A lógica do Jogo

Recriando clássicos da história dos videogames

Casa do Código

MARCUS BECKER

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

Agradecimentos

Agradeo a toda minha famlia que, de alguma forma, em diversos momentos da minha vida, me ajudou a chegar ao lugar onde estou hoje.

Introdução

Este livro não busca somente recriar alguns grandes clássicos do mundo do entretenimento eletrônico, mas sim entender, de forma prática, a lógica envolvida em cada um deles, cujas mecânicas são utilizadas até hoje em muitos jogos modernos. Assim, dará ao leitor a base inicial necessária para a criação de seus próprios projetos.

Estrutura do livro

Contamos com 10 capítulos, sendo 3 deles dedicados aos protótipos de jogo, 6 para os jogos e 1 para revermos os desafios propostos nos outros capítulos.

Não só falamos de diversos jogos como abordamos, a cada novo jogo criado, aspectos gerais do desenvolvimento de jogos de forma gradual, desde a movimentação do personagem até o uso de sons e imagens. Quando terminar o último capítulo, você poderá desenvolver qualquer um dos jogos abordados de forma completa, com início, meio e fim.

No capítulo 1, veremos como criar janelas, desenhar, interatividade e animação, além de criarmos um protótipo de jogo e detecção de colisão.

No capítulo 2, criaremos nossa versão do Space Invaders, explorando mais a movimentação do jogador e dos inimigos (que é um exercício lógico à parte) com uma pequena animação. Adicionaremos também pontuação e aumento de dificuldade.

No capítulo 3, o Pong que criamos é para dois jogadores, então, usaremos teclado e mouse para controlar cada um. Aprenderemos a separar o jogo em cenários, e criaremos um menu simples para configuração do modo de jogo.

No capítulo 4, ao criar o Blockade (Jogo da cobrinha), focamos em utilizar níveis, também conhecidos como fases ou *levels*, tendo assim um jogo com inúmeras possibilidades.

No capítulo 5, veremos por que Tetris é o quebra-cabeças animado mais viciante já criado, além de dedicarmos uma parte especial para falarmos de música e efeitos sonoros, e como calcular o tamanho do desenho de acordo com o tamanho tela ou janela do jogo.

No capítulo 6, criaremos um protótipo do Pac-Man só para abordarmos a Inteligência Artificial utilizada na perseguição e fuga dos personagens do jogo. Também veremos um tipo de colisão mais complexa.

No capítulo 7, criaremos um Pac-Man mais parecido com o jogo original, utilizando não somente imagens, mas também Sprites e Tiles.

No capítulo 8, usaremos o jogo Nave Quebrada para abordamos a matemática do jogo Asteroids, sem falar de matemática veremos: rotação, escalonamento, transição e ângulos.

No capítulo 9, além de uma versão para um ou dois jogadores de Asteroids, veremos mais de rotação e ângulos além de animação de Sprites e melhorias no processamento do jogo.

No capítulo 10, reveremos os desafios deixados para o leitor e, para os desafios onde apenas indicar a solução não é o suficiente, criaremos exemplos práticos.

Convenções de código usadas no livro

O código Java utilizado no livro aparece destacado de duas formas: em blocos de código com formatação diferenciada, quebrados por explicações; ou nas linhas do texto com formatação mais simples, geralmente quando nos referirmos a nomes de métodos ou variáveis.

Nem sempre conseguimos manter a indentação de código desejada, devido à largura da página. Códigos que já foram explicados ou tenham pouca importância para contexto da explicação serão substituídos por `"..."`.

Ferramentas e código-fonte: Java, Eclipse e GitHub

Usamos o Java para desenvolver os jogos deste livro, e o motivo é que o

Java é mais que uma linguagem de programação, é uma plataforma de desenvolvimento, presente em diversos dispositivos e milhares de computadores.

Muito provavelmente você já tenha a máquina virtual Java (*Java Virtual Machine*, ou JVM) instalada em seu computador. Se quiser verificar ou baixar, acesse http://java.com/pt_BR/.

Utilizamos como Ambiente de Desenvolvimento Integrado (*Integrated Development Environment*, ou IDE) o Eclipse, mas você pode utilizar outras IDEs, como por exemplo, o NetBeans.

Usamos também um repositório público para os fontes deste livro, para que os leitores possam contribuir com seus próprios códigos, seja criando mais fases e melhorias ou novas versões dos jogos abordados.

Escolhemos o Git, um poderoso versionador de código. Usamos o GitHub, que é um repositório Git na web, onde seus usuários podem compartilhar seus códigos de maneira pública e privada (pago).

Para obter os fontes, basta acessar https://github.com/logicadojogo/fontes.

Você também pode participar do nosso grupo de discussão, em: http://forum.casadocodigo.com.br.

As próximas seções são para os leitores que não estão familiarizados com algumas das ferramentas utilizadas no livro.

Instalação do Java para desenvolvedores

Caso ainda não desenvolva utilizando Java e queira fazê-lo, recomendamos a instalação do Kit de Desenvolvimento (*Java Development Kit*, ou JDK). Vá em http://www.oracle.com/technetwork/java/javase/downloads/, e clique no ícone de download para baixar a última versão.

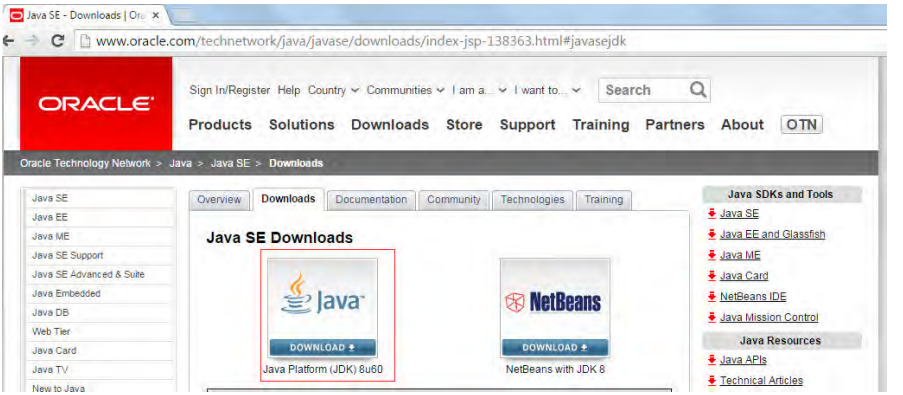

Fig. 1: Baixando JDK

Na próxima página, após aceitar o contrato de licença (*Accept License Agreement*), escolha a versão do seu sistema operacional:

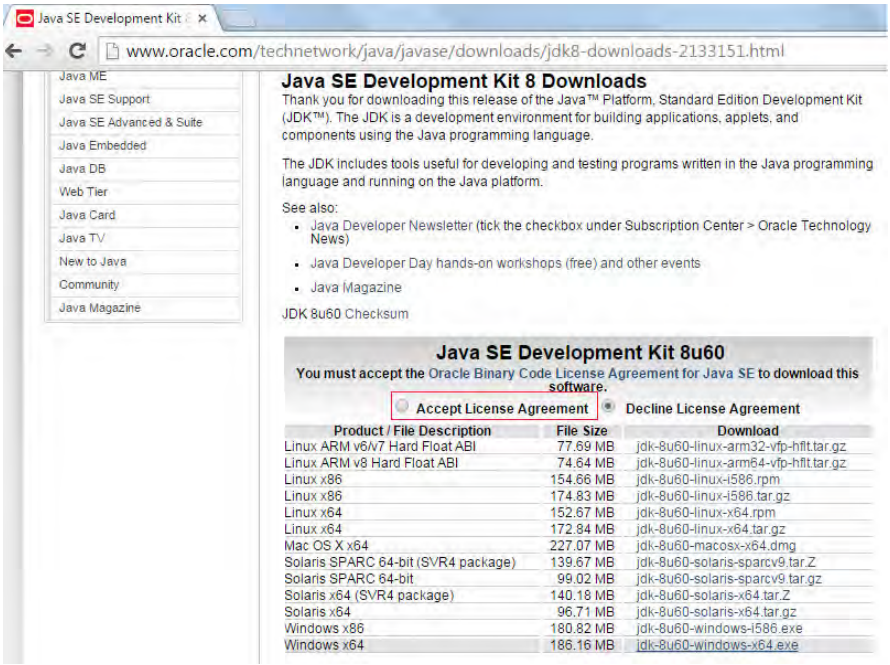

Fig. 2: Escolhendo a versão para seu Sistema Operacional

Siga as etapas de instalação padrão clicando em *Próximo* (*Next*)

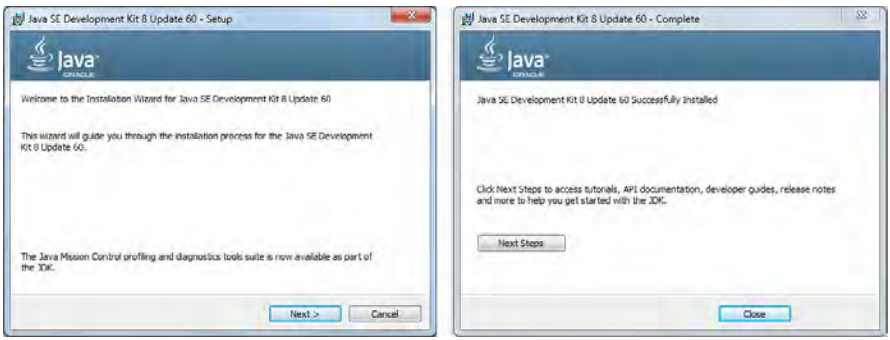

Fig. 3: Etapas da instalação

Ao final da instalação, clique em *Fechar* (*Close*). Uma vez com o Java ins-

talado, você pode compilar e executar os códigos deste livro utilizando alguns comandos, mas o faremos a partir do Eclipse.

Baixando o código-fonte do projeto

Em https://github.com/logicadojogo/fontes, você encontrará as duas principais versões do código divididas em duas *branches*.

Na *branch master*, temos o código principal explicado nos capítulos e, na *branch desafios*, temos o código com as modificações referentes aos desafios propostos no livro.

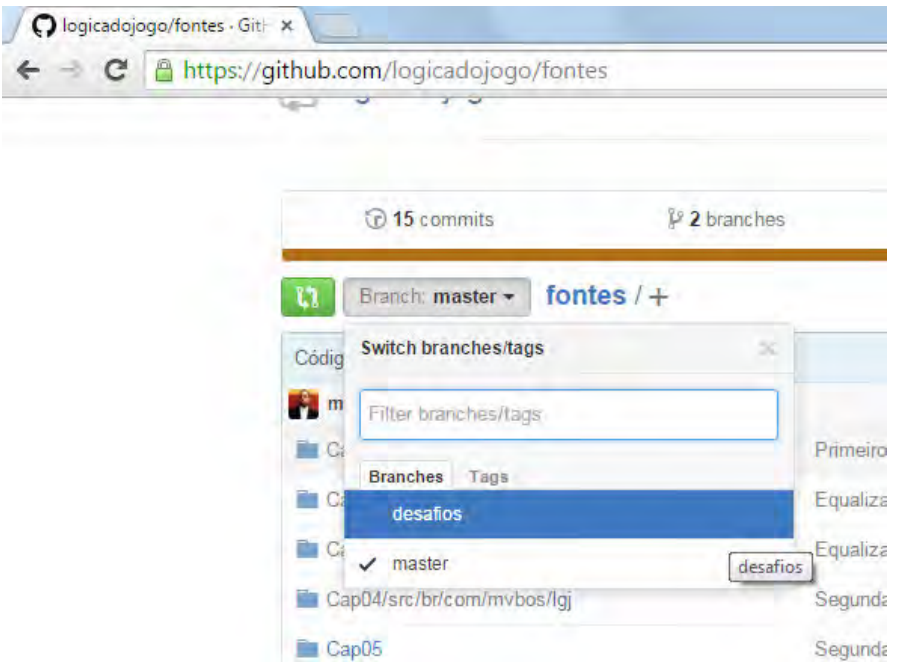

Fig. 4: Escolhendo a branch para download

Você não precisa ter uma conta cadastrada para acessar o código do livro, basta clicar em *Download ZIP*, no lado direito inferior da tela.

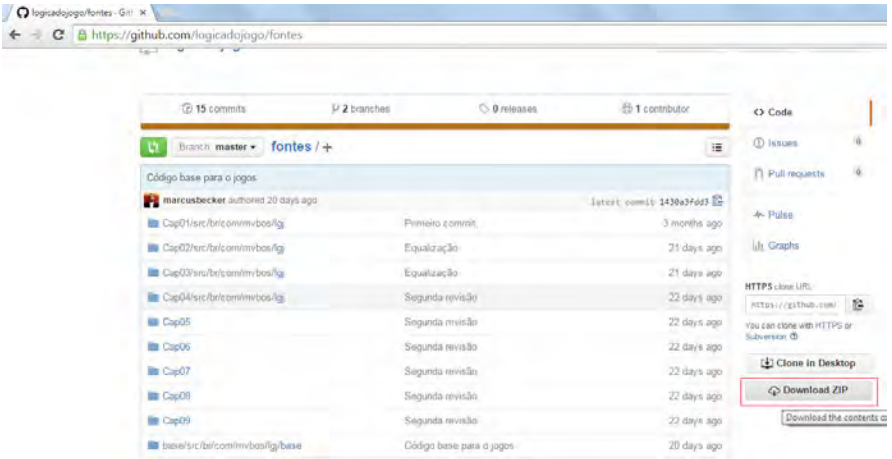

Fig. 5: Baixando Zip

Após baixar e extrair os arquivos, temos os códigos do projeto separados em pastas referentes aos capítulos do livro:

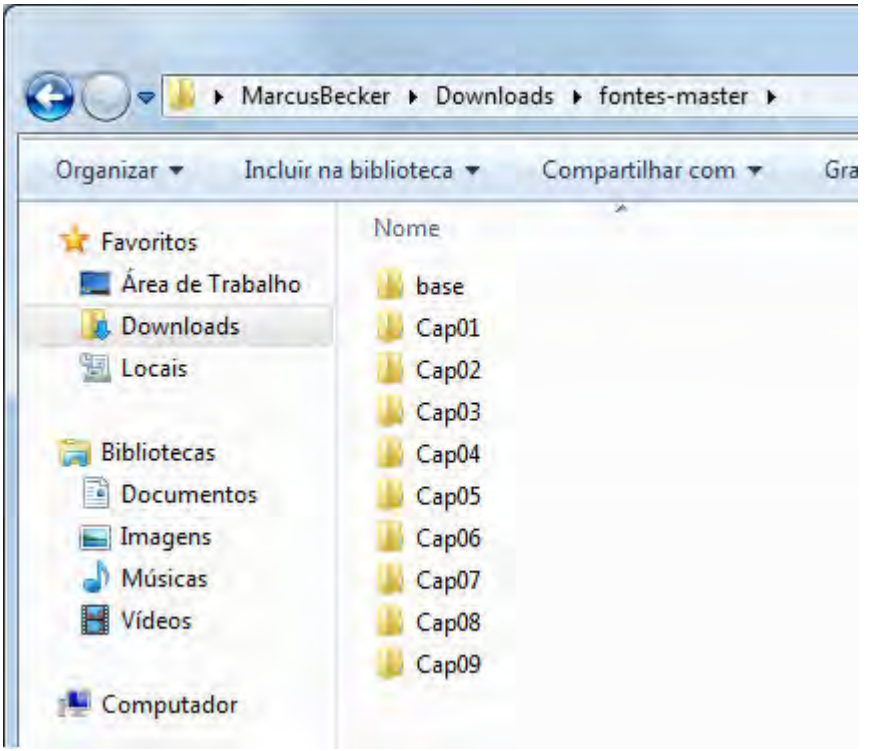

Fig. 6: Pastas separadas por capítulos

Depois disso, para você analisar, melhorar, personalizar ou simplesmente jogar os jogos desenvolvidos neste livro, só precisamos importar os códigos-fonte dos capítulos como um novo projeto no Eclipse.

Prefácio

Público-alvo

O principal público-alvo deste livro são desenvolvedores iniciantes independentes – não necessariamente desenvolvedores Java – que desejam aprender a criar seus próprios jogos, tendo como base os grandes clássicos: Space Invaders, Pong, Blockade, Tetris, Pac-Man e Asteroids.

Mas este livro também pode ser usado por outros interessados no mundo dos jogos, seja para conhecer um pouco mais sobre Java, a complexidade e lógica de cada jogo, para fazer porte do jogo para outra plataforma, para criar uma versão própria do código-fonte do livro, ou por simples curiosidade deste mundo tão desafiador.

Pré-requisitos

É muito importante ter conhecimento em lógica de programação e fundamentos de Orientação a Objetos (OO) para aproveitar completamente o conteúdo do livro, embora utilizemos o básico de OO e de recursos do Java, que estão presentes na maioria das linguagens de programação.

Se você já possuir conhecimento em Java, conseguirá tirar 100% de proveito deste livro. Já caso ainda não se sinta muito confortável com a linguagem, a editora Casa do Código possui um material para agradar desde iniciantes até velhos de guerra.

Sobre o autor

Marcus Becker Bacharel em Sistema de Informao, e trabalha com tecnologia desde 2005. Comeou com Flash, HTML e JavaScript, depois PHP, at que se apaixonou pelo Java a ponto de tirar certificao em 2009. J desenvolveu diversos pequenos jogos em diversas linguagens de programao, mas, desde 2012, vem se dedicando a criar jogos 2D com Java e Android, criando sua prpria Game Engine. Descubra mais em https://github.com/marcusbecker.

Sumário

1 Protótipo de jogo **1**

1.1 Desenhando na janela 2
1.2 Movendo as coisas pela tela 7
1.3 Interatividade 11
1.4 Objetivos e desafios 20
1.5 Caixa de ferramentas 27
1.6 Resumo 28

2 Invasão por Space Invaders **29**

2.1 Estrutura do jogo 31
2.2 Jogando o código 35
2.3 O disco chefe voador 42
2.4 Os invasores marcham 44
2.5 Codificando o jogo 50
2.6 Resumo 50

3 Meu nome é Pong, Ping Pong **51**

3.1 Separando a tela do cenário 52
3.2 Jogando o código 62
3.3 Codificando o jogo 74
3.4 Resumo 75

4 Jogo da cobrinha **77**
4.1 Níveis: aquilo que você faz para reaproveitar o cenário 78
4.2 Jogando o código 85
4.3 Codificando o jogo 92
4.4 Resumo 93

5 Tetris ou Quatro Quadrados **95**
5.1 Um jogo de arrays 96
5.2 Desenhando arrays 102
5.3 Jogando o código 105
5.4 Efeitos sonoros 117
5.5 Tetris sonoro 120
5.6 Programar, desenhar e ainda ter de compor? 124
5.7 Codificando o jogo 125
5.8 Resumo 125

6 Pac-Man, vulgo Come-come **127**
6.1 Caça fantasma 129
6.2 Jogando o código 141
6.3 Um pouco de I.A. não faz mal 151
6.4 Codificando o jogo 158
6.5 Resumo 158

7 Come-come e os bitmaps **161**
7.1 Uma imagem vale mais do que mil linhas de código 163
7.2 Jogando o código 172
7.3 Codificando o jogo 185
7.4 Resumo 186

8 Um jogo de ângulos e rotações **187**
8.1 Rotação 188
8.2 Escalonamento e transição 195
8.3 Um jogo para girar e atirar 198

8.4 Jogando o código 205
8.5 Codificando o jogo 213
8.6 Resumo 215

9 Asteroids: o jogo que sabe representar a física 217
9.1 Um é bom, dois é multiplayer 218
9.2 Dividir para destruir 229
9.3 Jogando o código 233
9.4 Codificando o jogo 251
9.5 Resumo 253

10 Última fase, último chefe e próximos jogos 255
10.1 Desafios 256
10.2 Use a caixa de ferramentas para criar seus próprios jogos . . 271

11 Referências bibliográficas 273

Capítulo 1

Protótipo de jogo

Neste capítulo, criaremos a base para os jogos que serão desenvolvidos nos capítulos seguintes. A cada novo capítulo e jogo criado, aumentaremos nossa biblioteca de código.

Veremos nas próximas seções, código após código, como criar nosso protótipo de jogo passando pelas etapas:

- Desenho;
- Animação;
- Interação;
- Objetivos;
- Desafios.

1.1 Desenhando na janela

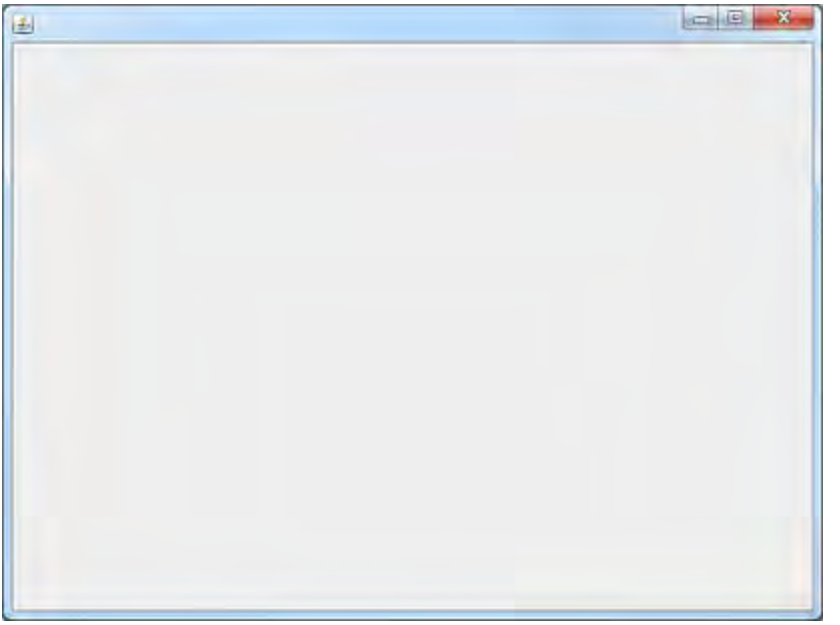

Fig. 1.1: Primeira janela

Gosto de pensar nos jogos eletrônicos como "uma animação interativa com objetivos e desafios". Então, vamos dividir o desenvolvimento do jogo base assim. Para animar, precisamos desenhar e, para desenhar, precisamos de uma tela; e a tela, por sua vez, precisa de uma janela/moldura. Para criar uma janela em Java é simples, faça assim:

```
package br.com.mvbos.lgj;
import javax.swing.JFrame;

public class Janela extends JFrame {

    public Janela() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(640, 480);
        setVisible(true);
```

```
    }

    public static void main(String[] args) {
        new Janela();
    }
}
```

Criamos uma classe chamada `Janela` que estende `JFrame`, que você pode pensar como uma moldura. Depois, no método construtor, definimos a ação padrão ao clicar no botão fechar da janela. Logo abaixo, definimos o tamanho (640 de largura por 800 de altura), e exibimos a janela chamando o método `setVisible(true)`.

Se a janela (`JFrame`) é nossa moldura, nossa tela de pintura será o `JPanel`. Mas o `JPanel` não será uma boa tela de pintura para jogos a menos que modifiquemos alguns de seus comportamentos padrão. Por isso, em vez de instanciarmos um objeto, criaremos uma classe que estenda `JPanel` e, assim, teremos maior controle sobre ela.

Usaremos o atalho que Java nos fornece, armazenando na nossa variável `tela` a instância da classe sobrescrita, dessa forma:

```
Package br.com.mvbos.lgj;

import java.awt.Graphics;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Janela extends JFrame {
    private JPanel tela;

    public Janela() {

        tela = new JPanel() {
            @Override
            public void paintComponent(Graphics g) {
                // A pintura ocorre aqui
            }
        };
```

```
        super.getContentPane().add(tela);

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(640, 480);
        setVisible(true);
    }

    public static void main(String[] args) {
        new Janela();
    }
}
```

> **Dica**
>
> Não confundir `paintComponent` com `paintComponents`, use a versão no singular.

Assim, podemos personalizar o método `paintComponent`, que para facilitar nossa vida, nos fornece o objeto `Graphics`, e você pode pensar nele como sendo nosso pincel. Depois de criada a tela, basta adicioná-la na moldura chamando `super.getContentPane().add(tela)`.

Temos a moldura (`JFrame`), a tela (`JPanel`) e o pincel (`Graphics`), com isso já podemos desenhar:

```
package br.com.mvbos.lgj;
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class AnimaTelaDesenho extends JFrame {

    private JPanel tela;

    public AnimaTelaDesenho() {
        tela = new JPanel() {
            @Override
```

```
            public void paintComponent(Graphics g) {
                g.setColor(Color.BLUE);
                g.drawLine(0, 240, 640, 240);
                g.drawRect(10, 25, 20, 20);
                g.drawOval(30, 20, 40, 30);

                g.setColor(Color.YELLOW);
                g.drawLine(320, 0, 320, 480);
                g.fillRect(110, 125, 120, 120);
                g.fillOval(230, 220, 240, 230);

                g.setColor(Color.RED);
                g.drawString("Eu seria um ótimo Score!", 5, 10);
            }
        };

        getContentPane().add(tela);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(640, 480);
        setVisible(true);
        tela.repaint();
    }

    public static void main(String[] args) {
        new AnimaTelaDesenho();
    }
}
```

Dentro do método `paintComponent`, antes da chamada para o método de desenho, definimos a cor com a qual esse desenho será pintado (azul). Desenhamos uma linha reta (`g.drawLine`) na horizontal, um quadrado (`g.drawRect`) e uma oval (`g.drawOval`) sem preenchimento. Sendo que os dois primeiros parâmetros são referentes ao posicionamento na tela, eixo X e Y. Os dois últimos são referentes ao tamanho do desenho.

Logo abaixo, depois de definirmos a cor amarela, desenhamos uma linha centralizada na vertical, com nosso quadrado e um oval preenchidos (`fillRect` e `fillOval`). Na cor vermelha, utilizamos `g.drawString` para "desenharmos uma frase".

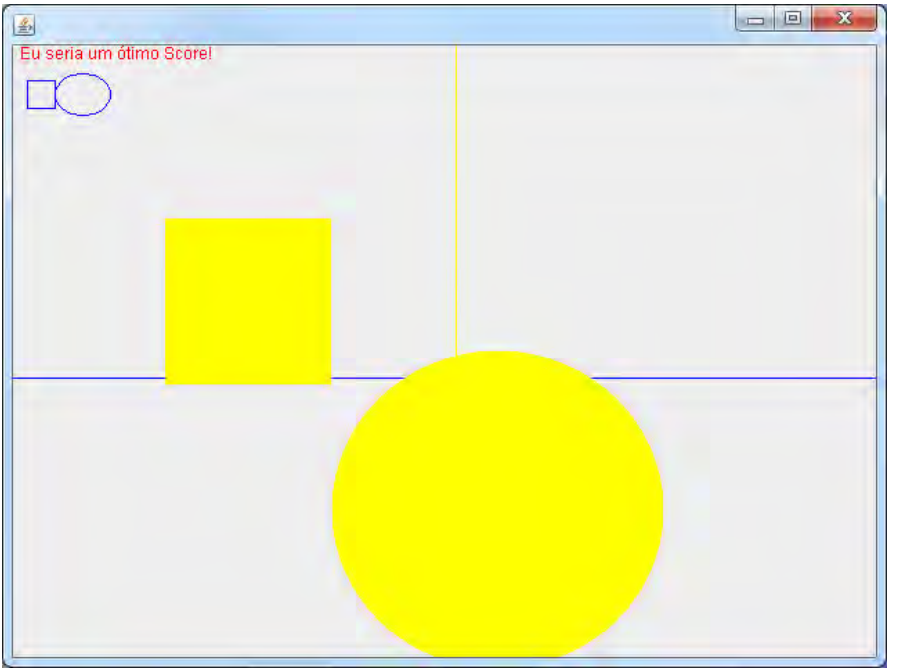

Fig. 1.2: Primeiro desenho

Dica:

Lembre-se, os métodos que começam com `draw` são sem preenchimento, e os que começam com `fill` são preenchidos.

Estas são apenas algumas funções de desenho e, nos próximos capítulos, vamos explorá-las melhor. Antes de utilizarmos as funções de desenho, setamos a cor do pincel usando as constantes de cores do Java. Você ainda pode usar sua própria cor criando um novo objeto `Color`, passando como parâmetro as tonalidades RGB (*Red* = vermelho; *Green* = verde; e *Blue* = Azul), dessa forma: `Color azulClaro = new Color(135, 206, 250);`.

As funções de desenho vetorial têm uma equivalente para desenho preenchido (começam com `fill`) ou apenas borda (começam com `draw`). Os

dois primeiros parâmetros são as posições de origem dos eixos X e Y em relação à tela, os dois últimos são a altura e largura ou, no caso das retas, a posição final dos eixos X e Y, ambos com medição em pixel.

Então, para desenhar uma linha vertical (`g.drawLine`) reta no meio da tela, definimos que ela iniciará em 320 do eixo X, sendo que nossa tela tem o tamanho de 640, e 0 no eixo Y. Seu eixo X final terá o mesmo valor inicial; caso contrário, não seria uma linha reta, e seu eixo Y final será a altura da tela, nesse caso, 480.

Para desenhar um retângulo (quadrado se tiver todos os lados iguais) preenchido (`g.fillRect`), definimos para os eixos X e Y, 110 e 125, com a largura e altura de 120 pixels. Com a chamada `g.drawString`, escrevemos um texto na tela utilizando tamanho e fonte padrão, sendo que o eixo Y (segundo parâmetro) representa a linha de base do texto.

1.2 Movendo as coisas pela tela

A animação tem a mesma base desde que foi criada. As imagens são mostradas em quadros (*frames*) que mudam a cada *n* segundos. Agora que temos nossa primeira imagem, vamos gerar novas imagens e mostrá-las em longos 20 frames por segundo, considerando que muitos jogos rodam a 60 FPS.

```
...
public class AnimacaoTela extends JFrame {
    private JPanel tela;
    private int fps = 1000 / 20; // 50
    private int ct; //contador
    private boolean anima = true;

    public void iniciaAnimacao() {
        long prxAtualizacao = 0;

        while (anima) {
            if (System.currentTimeMillis() >= prxAtualizacao) {
               ct++;
               tela.repaint();

               prxAtualizacao = System.currentTimeMillis() + fps;
```

```
                if (ct == 100)
                    anima = false;
            }
        }
    }
...
```

Como estamos trabalhando com milissegundos, nossa variável `fps` recebe o valor 50 (1000 / 20). A variável `ct` (nosso contador) é responsável pela mudança na tela, seja alterando os eixos ou o tamanho dos desenhos, enquanto `anima` encerrará a animação (o *loop*) quando o contador chegar a 100. Esse é o núcleo básico de um jogo, um laço (*loop*) infinito que faz atualizações e pinturas diversas vezes por segundo.

Dentro do método `iniciaAnimacao`, usamos `System.currentTimeMillis` para obter o tempo corrente em milésimos de segundo, e quando esse tempo for maior ou igual ao valor armazenado na variável `prxAtualizacao`, atualizamos o contador e pintamos a tela novamente.

Depois, passamos o valor atualizado para executarmos a próxima atualização e, assim, mantemos uma constância de frames por segundo até que o contador chegue a 100. Manter a constância de tempo é muito importante, e existem técnicas sofisticadas para isso, veremos um pouco mais no capítulo 9.

> **Dica**
>
> Lembre-se de pegar o tempo corrente novamente após a atualização e pintura da tela.

```
...
    public AnimacaoTela() {
        tela = new JPanel() {
            @Override
            public void paintComponent(Graphics g) {
                // Limpando os desenhos anteriores
```

```
                g.setColor(Color.WHITE);
                g.fillRect(0, 0, tela.getWidth(),
                           tela.getHeight());

                g.setColor(Color.BLUE);
                g.drawLine(0, 240 + ct, 640, 240 + ct);
                g.drawRect(10, 25 + ct, 20, 20);
                g.drawOval(30 + ct, 20, 40, 30);

                g.setColor(Color.YELLOW);
                g.drawLine(320 - ct, 0, 320 - ct, 480);
                g.fillRect(110, 125, 120 - ct, 120 - ct);
                g.fillOval(230, 220, 240 + ct, 230);

                g.setColor(Color.RED);
                g.drawString("Eu seria um ótimo Score! " +
                             ct, 5, 10);
            }
        };

        getContentPane().add(tela);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(640, 480);
        setVisible(true);
        tela.repaint();
    }

    public static void main(String[] args) {
        AnimacaoTela anima = new AnimacaoTela();
        anima.iniciaAnimacao();
    }
}
```

No método `paintComponent`, além de utilizarmos a variável `ct` para modificar as formas dos desenhos, a outra alteração é que agora apagamos o desenho anterior, desenhando um retângulo branco que preenche toda a tela. É como se virássemos a página antes do próximo desenho.

Se você já estiver preocupado com performance, saiba que além de o Java ser um desenhista performático, para evitar a sensação de atraso durante a

repintura da tela, usaremos uma segunda tela de pintura, em outras palavras, um *buffer*. O resultado da nossa animação será:

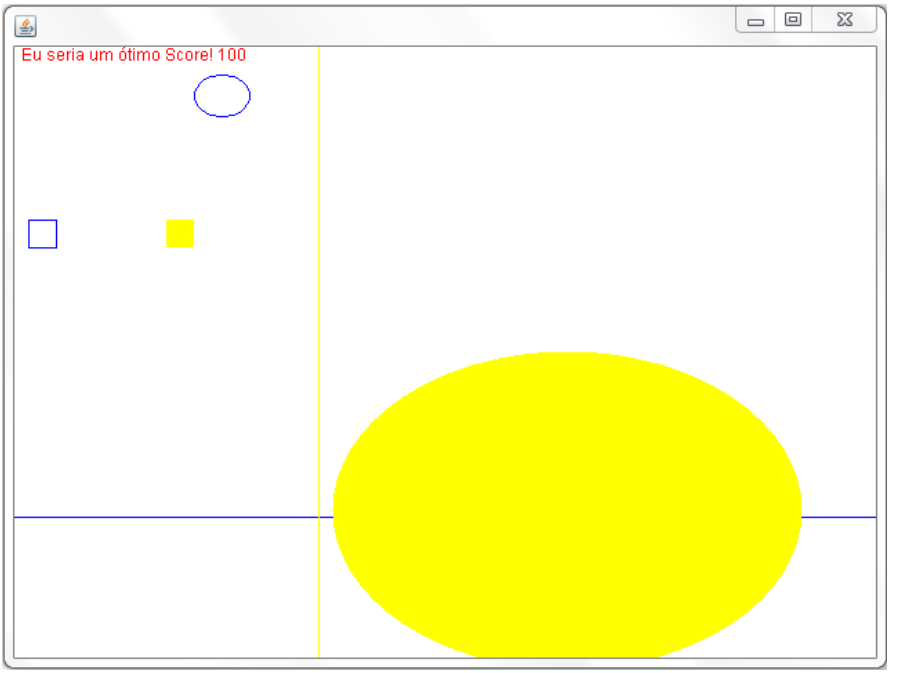

Fig. 1.3: Primeira animação

Assim como a classe `Graphics2D` do Java, durante o livro utilizamos o sistema de coordenadas X, Y, seja para posicionar a janela ou o desenho dentro dela. Esse sistema utiliza pixels (que são valores inteiros) como unidade de medida, então a posição `0, 0` está no canto superior esquerdo, sendo que o eixo x aumenta para direita e o eixo y para baixo, podendo ser negativo ou maior que o tamanho do frame (neste caso, não sendo visíveis no desenho).

No exemplo anterior, criamos uma janela de 640 pixels de largura por 480 pixels de altura, então seu canto superior esquerdo é `0, 0` e seu canto inferior direito `640, 480`. Tendo dito isso, vale adiantar que a medida de velocidade que usamos nos jogos é a de pixels por frame, ou *Pixels per frame* (PPF).

1.3 Interatividade

Nos jogos de hoje, até *cutscene* tem interatividade. A interatividade consiste em obter a entrada do jogador e refleti-la no jogo de alguma forma. Neste livro, nos limitaremos a teclado e mouse, não que interagir por meio de sons capturados do microfone ou imagens da webcam, ou um simples joystick, não seja tentador, mas infelizmente foge do escopo do livro.

O Java nos permite registrar os eventos que queremos monitorar, como um clique de mouse e uma tecla pressionada. No caso do clique do mouse, o evento é registrado diretamente na tela de pintura (`JPanel`). A janela (`JFrame`) fica responsável pelas entradas do teclado. No exemplo a seguir, usamos as setas do teclado para mover um quadrado azul pela tela:

```
package br.com.mvbos.lgj;

import java.awt.*;
import javax.swing.*;

import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;

public class Interativo extends JFrame {
    private JPanel tela;
    private int px;
    private int py;
    private boolean jogando = true;

    private final int FPS = 1000 / 20; // 50

    public void inicia() {
        long prxAtualizacao = 0;
        while (jogando) {
            if (System.currentTimeMillis() >= prxAtualizacao) {
               tela.repaint();
               prxAtualizacao = System.currentTimeMillis() + FPS;
            }
        }
    }
```

Começamos inserindo dois novos `imports` do pacote `java.awt.event`: `KeyEvent` e `KeyListener` e, na criação da janela, adicionamos nosso ouvinte do teclado com `super.addKeyListener`.

No método construtor da classe, monitoramos apenas o evento de tecla pressionada (`keyPressed`) e, quando ele ocorre, atualizamos as variáveis de `px` (posição no eixo X) e `py` (posição no eixo Y).

```
public Interativo() {
    super.addKeyListener(new KeyListener() {

        @Override
        //Evento para tecla apertada
        public void keyTyped(KeyEvent e) {
        }

        @Override
        //Evento para tecla liberada
        public void keyReleased(KeyEvent e) {
        }

        @Override
        //Evento para tecla pressionada
        public void keyPressed(KeyEvent e) {
            int tecla = e.getKeyCode();
            switch (tecla) {
            case KeyEvent.VK_ESCAPE:
                // Tecla ESC
                jogando = false;
                dispose(); // para fechar a janela
                break;
            case KeyEvent.VK_UP:
                // Seta para cima
                py--;
                break;
            case KeyEvent.VK_DOWN:
                // Seta para baixo
                py++;
                break;
            case KeyEvent.VK_LEFT:
```

```
                // Seta para esquerda
                px--;
                break;
            case KeyEvent.VK_RIGHT:
                // Seta para direita
                px++;
                break;
            }
        }
    });
```

Por meio da interface `KeyListener`, conseguimos tomar ações de acordo com o evento ocorrido, e utilizamos as constantes da classe `KeyEvent` para não termos de nos preocupar ou tentar descobrir o código referente à tecla pressionada.

```
tela = new JPanel() {
            @Override
            public void paintComponent(Graphics g) {
                g.setColor(Color.WHITE);
                g.fillRect(0, 0,
                    tela.getWidth(), tela.getHeight());

                int x = tela.getWidth() / 2 - 20 + px;
                int y = tela.getHeight() / 2 - 20 + py;

                g.setColor(Color.BLUE);
                g.fillRect(x, y, 40, 40);
                g.drawString("Agora eu estou em "
                                + x + "x" + y, 5, 10);
            }
        };

        getContentPane().add(tela);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(640, 480);
        setVisible(true);
    }
```

```
    public static void main(String[] args) {
        Interativo jogo = new Interativo();
        jogo.inicia();
    }
}
```

Para desenhar o quadrado azul no centro da tela, obtemos metade do tamanho da tela subtraindo a metade do tamanho do quadrado, neste caso, 20 pixels. Embora a janela tenha o tamanho de 640, metade da tela é menor que 320, já que devemos considerar as bordas e a barra de título da janela. Quando pressionado para cima, decrementamos `px`, fazendo o desenho subir; caso contrário, incrementamos `px`. O mesmo ocorre com `py`, movendo o desenho para esquerda (negativo) ou direita (positivo). Outra tecla mapeada é a tecla `ESC`, que pode ser usada como pausa, abrir menus ou encerrar o jogo.

> **Dica**
>
> O evento `keyTyped` não reconhece as setas do teclado.

Vale notar que, em Java, os ouvintes são monitorados em uma `Thread` separada da `Thread` principal. Então, quando você pressiona a tecla seta para cima, por exemplo, a variável `py` está sendo atualiza fora do tempo do desenho de 50ms, deixando o jogo inconstante. Outro problema é que, da forma atual, não conseguimos trabalhar com duas teclas pressionadas ao mesmo tempo. Para alguns jogos, isso não fará diferença, para outros, trará problemas de jogabilidade fazendo você perder prêmios nessa categoria. A nova versão utiliza uma abordagem melhor:

```
package br.com.mvbos.lgj;
...
public class Interativo2 extends JFrame {
...
    private boolean[] controleTecla = new boolean[4];

    public Interativo2() {
```

```
        this.addKeyListener(new KeyListener() {

            @Override
            public void keyTyped(KeyEvent e) {
            }

            @Override
            public void keyReleased(KeyEvent e) {
                setaTecla(e.getKeyCode(), false);
            }

            @Override
            public void keyPressed(KeyEvent e) {
                setaTecla(e.getKeyCode(), true);
            }
        });

        tela = new JPanel() {

            @Override
            public void paintComponent(Graphics g) {
                ...
            }
        };
        ...
    }
```

Transferimos a responsabilidade de iniciar e parar a movimentação do nosso objeto para o método `setaTecla` que, além de receber o código da tecla pressionada (*pressed*) ou liberada (*released*), recebe também um valor booleano para indicar se estamos liberando (`false`) ou pressionando (`true`).

```
public void inicia() {
    long prxAtualizacao = 0;

    while (jogando) {
        if (System.currentTimeMillis() >= prxAtualizacao) {
           atualizaJogo();
```

```
            tela.repaint();

            prxAtualizacao = System.currentTimeMillis() + FPS;
        }
    }
}

private void atualizaJogo() {
    if (controleTecla[0])
        py--;
    else if (controleTecla[1])
        py++;

    if (controleTecla[2])
        px--;
    else if (controleTecla[3])
        px++;
}
```

Agora, antes de atualizarmos nossa `tela`, chamamos o método `atualizaJogo`, que atualiza as variáveis de acordo com os valores do nosso array `controleTecla`, controlado pelo nosso método `setaTecla` que ficou assim:

```
private void setaTecla(int tecla, boolean pressionada) {
        switch (tecla) {
        case KeyEvent.VK_ESCAPE:
            // Tecla ESC
            jogando = false;
            dispose();
            break;
        case KeyEvent.VK_UP:
            // Seta para cima
            controleTecla[0] = pressionada;
            break;
        case KeyEvent.VK_DOWN:
            // Seta para baixo
            controleTecla[1] = pressionada;
            break;
```

```
        case KeyEvent.VK_LEFT:
            // Seta para esquerda
            controleTecla[2] = pressionada;
            break;
        case KeyEvent.VK_RIGHT:
            // Seta para direita
            controleTecla[3] = pressionada;
            break;
        }
    }

    public static void main(String[] args) {
        Interativo2 jogo = new Interativo2();
        jogo.inicia();
    }
}
```

Comparado com a versão anterior, agora a movimentação do nosso quadrado azul ficou mais fluida e constante, além de podermos movimentá-lo na diagonal. Observe que, em vez de atualizarmos a imagem fora do tempo definido, apenas armazenamos as teclas que foram pressionadas e as que foram liberadas, utilizando a variável `controleTecla`. E, no tempo de cada frame, executamos a atualização do jogo, método `atualizaJogo()`, que cuida da movimentação do objeto pela tela.

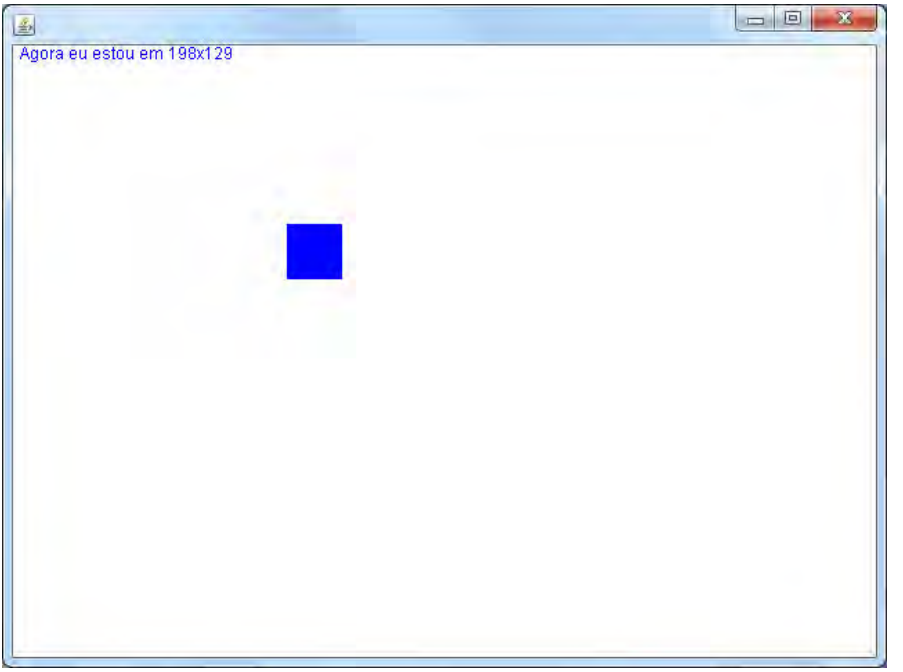

Fig. 1.4: Movendo objetos pela tela

Ouvir cliques do mouse é tão simples quanto, embora o evento de arrastar e soltar (*Drag and Drop*) exija um pouco mais de código. Lembre-se de que, em vez de registrarmos o evento para a janela, vamos registrar na tela (`JPanel`); caso contrário, teríamos de considerar o tamanho da borda e da barra de título, ao obter as posições e cliques.

```
package br.com.mvbos.lgj;
import java.awt.Point;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
...

public class InterativoMouse extends JFrame {
    private JPanel tela;
    private int px, py;
    private Point mouseClick = new Point();
```

```
    private boolean jogando = true;
    private final int FPS = 1000 / 20; // 50

    public InterativoMouse() {

        tela = new JPanel() {
            ...
        };

        tela.addMouseListener(new MouseListener() {

            @Override
            public void mouseReleased(MouseEvent e) {
                // Botão mouse liberado
            }

            @Override
            public void mousePressed(MouseEvent e) {
                // Botão mouse pressionado
            }

            @Override
            public void mouseExited(MouseEvent e) {
                // Mouse saiu da tela
            }

            @Override
            public void mouseEntered(MouseEvent e) {
                // Mouse entrou na tela
            }

            @Override
            public void mouseClicked(MouseEvent e) {
                // Clique do mouse
                mouseClick = e.getPoint();
            }
        });
        ...
    }
```

```
    public void inicia() {
        ...
    }

    private void atualizaJogo() {
        px = mouseClick.x;
        py = mouseClick.y;
    }

    public static void main(String[] args) {
        InterativoMouse jogo = new InterativoMouse();
        jogo.inicia();
    }
}
```

Assim como as teclas pressionadas, passamos os valores para a variável `mouseClick` que lemos na atualização do jogo. Não nos importamos com qual botão o clique é realizado, nem com os outros eventos, mas nos próximos capítulos, exploraremos mais sobre o assunto.

1.4 Objetivos e desafios

Um objetivo por si só já representa um desafio, mas nesse caso, desafios são pensados pelo desenvolvedor do jogo para gerar uma sensação de conquista (às vezes, ódio) no jogador. Particularmente falando, esta é uma etapa de grande prazer no desenvolvimento do jogo, elaborar desafios, pensando em como e de que forma outras pessoas vão superá-los. Com o básico que vimos até agora, vamos elaborar um pequeno desafio, que será impedir que quadrados azuis atravessem a linha cinza.

Nota

Aparentemente, os jogos ficaram tão fáceis com seus salvamentos automático e vida infinita, que um novo gênero de jogo surgiu, conhecido como *Roguelike*. Esse estilo de jogo preza pela dificuldade como parte do desafio do jogador, vide como exemplo, a série *Dark Souls*.

Fig. 1.5: Um jogo

O objetivo é fazê-lo pelo maior tempo possível, conseguindo a maior pontuação. Usaremos um código compacto, evitando conversões e padrões, mas não se acostume, é só dessa vez.

```
package br.com.mvbos.lgj;
```

```
//... Ocultamos os imports

public class UmJogo extends JFrame {

    private final int FPS = 1000 / 20; // 50

    class Elemento {
        public int x, y, largura, altura;
        public float velocidade;

        public Elemento(int x, int y, int width, int height) {
            this.x = x;
            this.y = y;
            this.largura = width;
            this.altura = height;
        }
    }

    private JPanel tela;
    private boolean jogando = true;
    private boolean fimDeJogo = false;

    private Elemento tiro;
    private Elemento jogador;
    private Elemento[] blocos;

    private int pontos;
    private int larg = 50; // Largura padrão
    private int linhaLimite = 350;
    private java.util.Random r = new java.util.Random();

    private boolean[] controleTecla = new boolean[4];
```

Definimos uma classe interna chamada `Elemento`, que contém as propriedades comuns aos três elementos do jogo (`blocos`, `jogador` e `tiro`). Declaramos algumas variáveis já conhecidas e algumas novas para controlar o jogo, que você não precisa se preocupar agora, já que veremos com mais detalhes em capítulos futuros.

```
public UmJogo() {

    this.addKeyListener(new KeyListener() {
    ...
    });

    tiro = new Elemento(0, 0, 1, 0);
    jogador = new Elemento(0, 0, larg, larg);
    jogador.velocidade = 5;

    blocos = new Elemento[5];
    for (int i = 0; i < blocos.length; i++) {
        int espaco = i * larg + 10 * (i + 1);
        blocos[i] = new Elemento(espaco, 0, larg, larg);
        blocos[i].velocidade = 1;
    }
```

No método construtor da classe, configuramos o tamanho do tiro (1 de largura, 0 de altura), a posição será relativa ao jogador. Tanto o jogador (quadrado inferior verde) quanto os obstáculos (quadrados superiores azuis) têm o mesmo tamanho, mas a velocidade deles é diferente: 5 para o jogador e 1 para os obstáculos, que são posicionados lado a lado com um espaçamento de 10 pixels entre eles.

```
tela = new JPanel() {

        private static final long serialVersionUID = 1L;

        @Override
        public void paintComponent(Graphics g) {
            g.setColor(Color.WHITE);
            g.fillRect(0, 0,
                    tela.getWidth(), tela.getHeight());

            g.setColor(Color.RED);
            g.fillRect(tiro.x, tiro.y,
                    tiro.largura, tela.getHeight());

            g.setColor(Color.GREEN);
```

```
            g.fillRect(jogador.x, jogador.y,
                    jogador.largura, jogador.altura);

            g.setColor(Color.BLUE);
            for (Elemento bloco : blocos) {
                g.fillRect(bloco.x, bloco.y,
                    bloco.largura, bloco.altura);
            }

            g.setColor(Color.GRAY);
            g.drawLine(0, linhaLimite,
                tela.getWidth(), linhaLimite);

            g.drawString("Pontos: " + pontos, 0, 10);
        }
    };

    getContentPane().add(tela);

    ...
    setResizable(false);

    jogador.x = tela.getWidth() / 2 - jogador.x / 2;
    jogador.y = tela.getHeight() - jogador.altura;
    tiro.altura = tela.getHeight() - jogador.altura;
}

public void inicia() {
...
}
```

Estamos utilizando os métodos já conhecidos para desenho, mas os valores estão no próprio objeto `Elemento`. Além de não deixarmos a tela ser redimensionada com `setResizable(false)`, definimos a posição inicial do `jogador` e do `tiro` usando como base o tamanho da `tela`.

No método `atualizaJogo`, movemos o jogador na horizontal conforme a tecla pressionada, sempre validando para que, se ele sair da tela, apareça do lado oposto, e posicionamos o tiro centralizado no jogador.

```
private void atualizaJogo() {
    if (fimDeJogo)
        return;

    if (controleTecla[2])
        jogador.x -= jogador.velocidade;
    else if (controleTecla[3])
        jogador.x += jogador.velocidade;

    if (jogador.x < 0)
        jogador.x = tela.getWidth() - jogador.largura;

    if (jogador.x + jogador.largura > tela.getWidth())
        jogador.x = 0;

    tiro.y = 0;
    tiro.x = jogador.x + jogador.largura / 2;
```

Para cada bloco, verificamos se algum passou totalmente da linha limite; caso aconteça, encerramos o jogo. Quando algum `bloco` colidir com o `tiro` (veremos melhor detecção de colisão a seguir), ele voltará gradualmente ao topo da tela com o dobro da velocidade de decida. Caso não ocorra a colisão, descemos o `bloco` utilizamos o fator `sorte`.

```
for (Elemento bloco : blocos) {

            if (bloco.y > linhaLimite) {
                fimDeJogo = true;
                break;
            }

            if (colide(bloco, tiro) && bloco.y > 0) {
                bloco.y -= bloco.velocidade * 2;
                tiro.y = bloco.y;

            } else {
                int sorte = r.nextInt(10);
                if (sorte == 0)
                    bloco.y += bloco.velocidade + 1;
```

```
                else if (sorte == 5)
                    bloco.y -= bloco.velocidade;
                else
                    bloco.y += bloco.velocidade;
            }
        }

        pontos = pontos + blocos.length;
    }

    private boolean colide(Elemento a, Elemento b) {
        if (a.x + a.largura >= b.x && a.x <= b.x + b.largura) {
            return true;
        }

        return false;
    }

    private void setaTecla(int tecla, boolean pressionada) {
        ...
    }

    public static void main(String[] args) {
        UmJogo jogo = new UmJogo();
        jogo.inicia();
    }

}
```

Mesmo para um protótipo, precisamos de bastante código. Temos código para pintar a tela, atualizar os objetos, receber e tratar a entrada do jogador, verificar colisões. E o tamanho do código só tende a crescer conforme vamos construindo novos jogos nos capítulos seguintes. Então, usaremos o poder da Orientação ao Objeto, além de reaproveitarmos partes comuns entre os jogos, criando e ampliando nossa caixa de ferramentas de código para fazemos mais jogos com menos trabalho.

1.5 Caixa de ferramentas

Nosso primeiro item da caixa de ferramentas é a classe `Util`, que vem com o método mais utilizado de todos os jogos, responsável pela detecção de colisão no jogo, mais precisamente colisão entre os objetos da classe `Elemento`.

> **Nota**
>
> Eu, como leitor, prefiro nomes curtos para variáveis, assim fica menos cansativo digitar o código de exemplo. Por isso, adotei essa abordagem para escrever o livro.

```
package br.com.mvbos.lgj.base;

public class Util {

    public static boolean colide(Elemento a, Elemento b) {
        if (!a.isAtivo() || !b.isAtivo())
            return false;
        //posição no eixo X + largura do elemento A e B
        final int plA = a.getPx() + a.getLargura();
        final int plB = b.getPx() + b.getLargura();

        //posição no eixo Y + altura do elemento A e B
        final int paA = a.getPy() + a.getAltura();
        final int paB = b.getPy() + b.getAltura();

        if (plA > b.getPx() && a.getPx() < plB
                    && paA > b.getPy() && a.getPy() < paB) {
            return true;
        }

        return false;
    }
}
```

Nos próximos jogos, alguns elementos podem fazer parte do jogo, mas não estar ativos,. Assim, por conveniência, colisões com elementos inativos

retornam falso.

Primeiramente, obtemos a posição mais largura do `Elemento a` (`pla`) e `Elemento b` (`plb`), como também a posição mais altura de ambos (`paA` e `paB`). Sendo que, para a largura, consideramos o eixo X e altura o eixo Y.

Depois disso, verificamos a colisão total (nos eixos X e Y), checando se a posição somada a largura do objeto A é maior a posição do objeto B, e se a posição do objeto A é menor a posição mais a largura do objeto B. Fazemos o mesmo processo para a altura. Se as duas primeiras validações derem verdadeiro, temos uma colisão no eixo X; se as duas últimas derem verdadeiro, uma colisão no eixo Y.

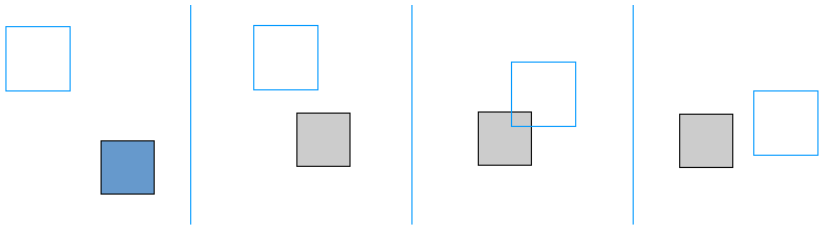

Fig. 1.6: Colisões

Na figura 1.6, os dois primeiros quadrados não estão colidindo; já no segundo quadro, eles colidem nos eixos X. No terceiro quadro, temos a colisão em ambos os eixos; por fim, no último, somente a colisão no eixo Y.

Criaremos jogos em que os objetos serão posicionados muito próximos um do outro, então, consideramos colisão apenas se a posição de um deles for maior, e não maior ou igual, a do outro elemento.

Vale lembrar que o código-fonte completo pode (e deve) ser baixado em https://github.com/logicadojogo/fontes.

1.6 Resumo

Neste capítulo, vimos alguns elementos básicos que compõem um jogo como: desenho, interatividade e animação. Criamos um protótipo de jogo e nossa função para detectar colisão. Estamos só nos aquecendo, que os jogos comecem. Aliás, que comecemos os jogos!

Capítulo 2

Invasão por Space Invaders

Space Invaders, o jogo que fez o Japão triplicar sua produção de moedas de 100-yen (eles não usavam fichas de fliperama), foi desenvolvido pela Taito Corporation, em 1978 no Japão, e lançado em outubro do mesmo ano nos Estados Unidos pela Midway. Neste capítulo, abordaremos com base em aspectos do jogo original:

- Movimentação do jogador e inimigos;
- Tiros na vertical;
- Pontuação;
- Aumento de dificuldade;
- Animação dos inimigos;

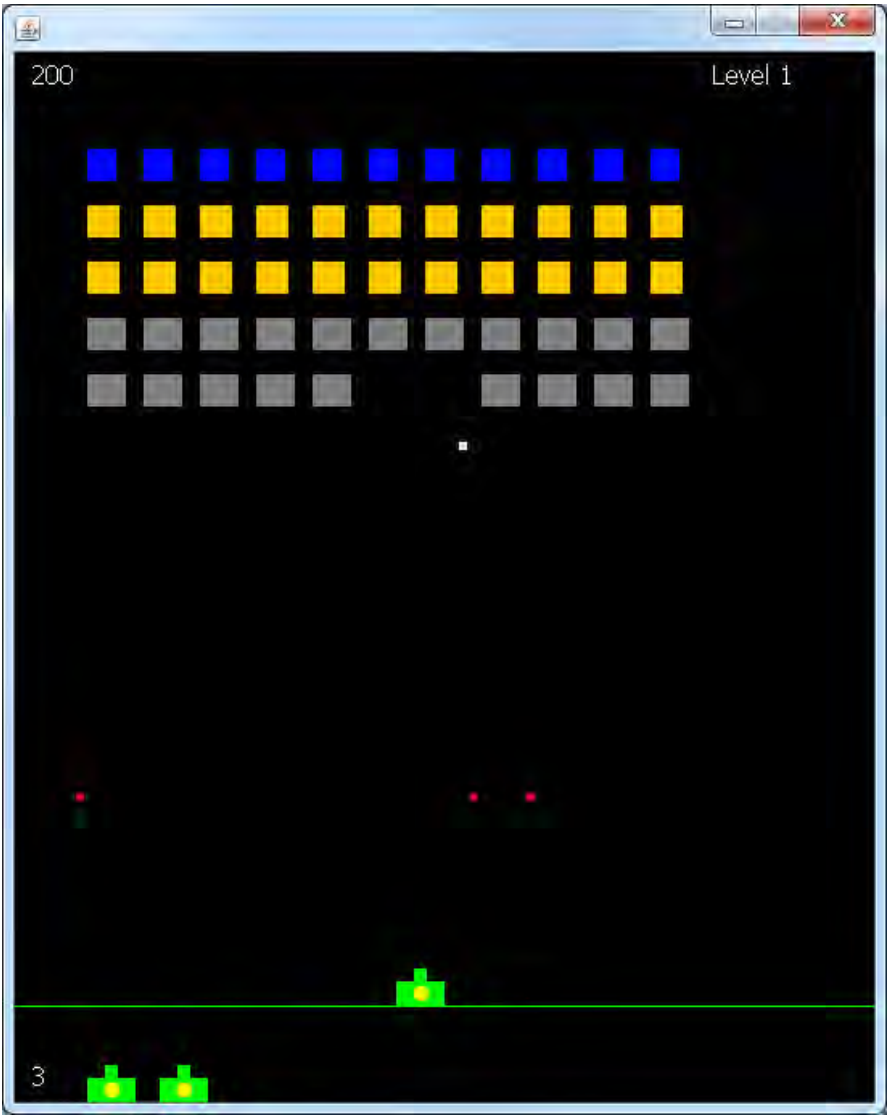

Fig. 2.1: Visualização do jogo

2.1 Estrutura do jogo

Um jogo é formado, por entre outras coisas, de elementos e quaisquer coisas que esses elementos sejam, por exemplo: naves, tanques, carros, pássaros zangados... todos eles têm aspectos em comum, como posição, tamanho e ações. Nosso tanque atira, nossa nave voa, e assim por diante. Por isso, para nossos jogos, teremos uma classe `Elemento` que inicialmente se parece com:

```
package br.com.mvbos.lgj.base;

import java.awt.Color;
import java.awt.Graphics2D;

public class Elemento {
    private int px;
    private int py;
    private int largura;
    private int altura;
    private int vel;
    private boolean ativo;
    private Color cor;

    public Elemento() {    }

    public Elemento(int px, int py, int largura, int altura) {
        this.px = px;
        this.py = py;
        this.largura = largura;
        this.altura = altura;
    }
```

Vimos algo parecido no capítulo anterior, mas vale relembrar que `px`, `py`, `largura` e `altura` são propriedades que utilizamos muito na hora de desenhar e detectar colisões com outros elementos do jogo. Para controlar a velocidade do personagem, usamos a propriedade `vel` e, se ele pode ser atualizado ou desenhado, utilizamos a propriedade `ativo`. E enquanto não trabalhamos com imagens, usaremos a propriedade `cor` para dar mais vida ao jogo.

```
public void atualiza() { }

public void desenha(Graphics2D g) {
    g.drawRect(px, py, largura, altura);
}

public void incPx(int x) { px = px + x; }
public void incPy(int y) { py = py + y; }
```

Um método para atualizar e outro para desenhar são dois métodos que todos nossos elementos devem ter. Por padrão, nossa classe `Elemento` não faz nenhuma atualização e desenha um retângulo sem preenchimento com base na posição e tamanho. Ela também vem de fábrica com dois métodos auxiliares para facilitar a mudança de posição do elemento na tela (`incPx()` e `incPy()`). Os outros métodos são para acessar os atributos privados da classe:

```
public int getLargura() {
        return largura;
    }

    public void setLargura(int largura) {
        this.largura = largura;
    }

    public int getAltura() {
        return altura;
    }

    public void setAltura(int altura) {
        this.altura = altura;
    }

    public int getPx() {
        return px;
    }

    public void setPx(int px) {
        this.px = px;
```

```
    }

    public int getPy() {
        return py;
    }

    public void setPy(int py) {
        this.py = py;
    }

    public int getVel() {
        return vel;
    }

    public void setVel(int vel) {
        this.vel = vel;
    }

    public boolean isAtivo() {
        return ativo;
    }

    public void setAtivo(boolean ativo) {
        this.ativo = ativo;
    }

    public Color getCor() {
        return cor;
    }

    public void setCor(Color cor) {
        this.cor = cor;
    }

}
```

Dica

No Eclipse, depois de declarar qualquer propriedade da classe, utilize o atalho: `ALT + SHIFT + S` e clique em `Generate Getters and Setters...` para gerar os métodos `get` e `set` delas.

Os principais objetos dos nossos jogos vão estender da classe `Elemento`, assim, além de reaproveitarmos os comportamentos e atributos em comum, muitos métodos precisarão simplesmente saber que nosso objeto é um `Elemento`, como por exemplo, o método `Colide` da classe `Util`.

Quando um elemento do jogo for muito genérico, ele será uma instância direta da classe `Elemento`, como `Elemento el = new Elemento()`. Quando precisarmos personalizar algum comportamento, instanciaremos dessa forma: `Elemento tanque = new Tanque()`. Caso nossa implementação adicione novos comportamentos, a declaração será a mesma da classe que instanciarmos: `Invader chefe = new Invader()`.

A estrutura do nosso projeto ficou conforme a figura:

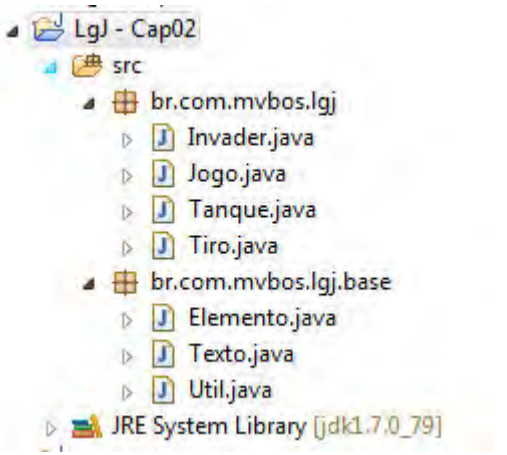

Fig. 2.2: Estrutura do projeto

2.2 Jogando o código

As naves inimigas estão alinhadas e em perfeita sincronia, indo da esquerda para a direita e avançando para a parte inferior da tela, conforme colidem com as laterais. Quanto mais naves derrubar, mais rápido será o avanço das restantes, até que a última pareça o *The Flash* em dia de corrida. Enquanto a única nave (neste jogo, tanque) defende a Terra com movimentos na horizontal e um tiro por vez.

Temos três objetos principais, `Invader`, `Tanque` e `Tiro`, que estendem da classe `Elemento` e modificam ou acrescentam novos comportamentos. Chamamos a classe de `Invader` em homenagem ao jogo, estes serão os inimigos invasores.

Para gerenciar as naves inimigas, criamos uma matriz multidimensional (array de arrays), 11 por 5, tendo um total de 55 inimigos mais o chefe, que também é um objeto `Invader`, mas se move somente na horizontal.

```
...
private Invader[][] invasores = new Invader[11][5];

private Invader.Tipos[] tipoPorLinha = { Tipos.PEQUENO,
    Tipos.MEDIO, Tipos.MEDIO, Tipos.GRANDE, Tipos.GRANDE };

...

private void carregarJogo() {
...
    for (int i = 0; i < invasores.length; i++) {
        for (int j = 0; j < invasores[i].length; j++) {
            Invader e = new Invader(tipoPorLinha[j]);

            e.setAtivo(true);

            e.setPx(i * e.getLargura() + (i + 1) * espacamento);
            e.setPy(j * e.getAltura() + j * espacamento +
                    linhaBase);

            invasores[i][j] = e;
        }
```

```
    }
...
}
```

Cada linha tem um tipo de inimigo, usamos o array `tipoPorLinha`, que armazena os valores da nossa enumeração (`enum`, que a grosso modo são constantes com mais recursos) definidos na classe `Invader` para controlar isso.

Os inimigos são posicionados um ao lado do outro e o posicionamento horizontal leva em conta a largura de cada mais um espaçamento adicional, por exemplo, considerando que o inimigo ocupe 20 pixels e o espaçamento seja de 15, o primeiro `Invader` começará na posição 15 (0 * 20 + 1 * 15), e o segundo na posição 50 (1 * 20 + 2 * 15), assim por diante.

O mesmo é feito para posicioná-los em linha, mas, neste caso, utilizamos um espaço maior (60 pixels), deixando espaço para escrevemos a pontuação e o level no topo da tela.

Utilizamos um contador para controlar a marcha dos invasores. Quanto menor for o número de inimigos, menor será o tempo de espera para a movimentação deles:

```
...
// Parte do código dentro do loop do jogo
if (contador > contadorEspera) {
    moverInimigos = true;
    contador = 0;
    contadorEspera = totalInimigos - destruidos - level * level;

} else {
    contador++;
}
```

A cada frame (loop do jogo), incrementamos a variável `contador` e, quando ela for maior que `contadorEspera`, que simplesmente é o total de inimigos subtraindo os inimigos destruídos e o dobro do level, movemos os invasores. Então, a cada avanço de nível, o jogo fica mais difícil.

```
if (tanque.isAtivo()) {
    if (controleTecla[2]) {
```

```
            tanque.setPx(tanque.getPx() - tanque.getVel());
        } else if (controleTecla[3]) {
            tanque.setPx(tanque.getPx() + tanque.getVel());
        }
    }

    // Pressionou espaço, adiciona tiro
    if (controleTecla[4] && !tiroTanque.isAtivo()) {
        tiroTanque.setPx(tanque.getPx() + tanque.getLargura() / 2
                            - tiroTanque.getLargura() / 2);
        tiroTanque.setPy(tanque.getPy() - tiroTanque.getAltura());
        tiroTanque.setAtivo(true);
    }

    if (chefe.isAtivo()) {
        chefe.incPx(tanque.getVel() - 1);

        if (!tiroChefe.isAtivo() && Util.colideX(chefe, tanque)) {
            addTiroInimigo(chefe, tiroChefe);
        }

        if (chefe.getPx() > tela.getWidth()) {
            chefe.setAtivo(false);
        }
    }
```

Se nosso `tanque` estiver ativo, realizamos o movimento para a esquerda ou direita caso as teclas correspondentes sejam pressionadas. O tiro do tanque e do chefe (nave que percorre a horizontal superior) são disparos em linha reta na vertical, sendo que o tiro do tanque parte de baixo para cima quando o jogador pressiona a tecla espaço, e o tiro do chefe de cima para baixo, quando ocorre uma colisão no eixo X.

Quando a posição do chefe for maior que a largura da tela, ele se torna inativo. Veremos mais sobre posicionamento logo a seguir.

Para os tiros saírem primeiramente das naves inferiores, percorremos os array de baixo para cima (do maior para o menor):

```
boolean colideBordas = false;
```

```
// Percorrendo primeiro as linhas, de baixo para cima
for (int j = invasores[0].length - 1; j >= 0; j--) {

    // Depois as colunas
    for (int i = 0; i < invasores.length; i++) {

        Invader inv = invasores[i][j];

        if (!inv.isAtivo()) {
            continue;
        }

        if (Util.colide(tiroTanque, inv)) {
            inv.setAtivo(false);
            tiroTanque.setAtivo(false);
            destruidos++;
            pontos = pontos + inv.getPremio() * level;
            continue;
        }
```

Se o invasor não estiver mais ativo (foi destruído), passamos para o próximo. Se ele estava ativo e foi atingido pelo tiro do nosso tanque, inativamos ambos, atualizamos a contagem de inimigos destruídos e a aumentamos a pontuação do jogador, levando em conta o tipo do invasor e o *level* do jogo.

```
if (moverInimigos) {

    inv.atualiza();

    if (novaLinha) {
        inv.setPy(inv.getPy() + inv.getAltura() +
                  espacamento);
    } else {
        inv.incPx(espacamento * dir);
    }

    if (!novaLinha && !colideBordas) {
        int pxEsq = inv.getPx() - espacamento;
        int pxDir = inv.getPx() + inv.getLargura() +
```

```
                        espacamento;

        if (pxEsq <= 0 || pxDir >= tela.getWidth())
            colideBordas = true;
    }
```

Quando chega a hora de mover os inimigos, verificamos se o movimento será na vertical (nova linha) ou na horizontal, para esquerda ou direita dependendo do valor de `dir` (1 ou -1). Caso não seja uma nova linha e a colisão com as bordas da tela ainda não tenha sido detectada, verificamos se o próximo movimento resultará em uma colisão.

Se todas as naves ficassem em jogo o tempo todo, precisaríamos verificar apenas a primeira e a última coluna, mas como o jogador pode eliminar colunas inteiras, precisamos fazer a verificação individual.

Os inimigos atiram e atiram bem, são apenas três tiros, mas há certa IA (Inteligência Artificial) neles (falaremos de IA no capítulo 6. O primeiro tiro é sempre do lardo esquerdo do jogador, o segundo na direção do jogador e o terceiro do lado direito.

```
if (!tiros[0].isAtivo() && inv.getPx() < tanque.getPx()) {
    addTiroInimigo(inv, tiros[0]);

} else if (!tiros[1].isAtivo() &&
      inv.getPx() > tanque.getPx() &&
      inv.getPx() < tanque.getPx() + tanque.getLargura()) {
    addTiroInimigo(inv, tiros[1]);

} else if (!tiros[2].isAtivo() &&
      inv.getPx() > tanque.getPx()) {
    addTiroInimigo(inv, tiros[2]);

}

if (!chefe.isAtivo() && rand.nextInt(500) == destruidos) {
    chefe.setPx(0);
    chefe.setAtivo(true);
}
```

Toda verificação utiliza a posição dos elementos no eixo X. Agora, só precisamos fechar as chaves que deixamos abertas.

```
}//Fim if moverInimigos

        // Desenhe aqui se quiser economizar no loop.
        // e.desenha(g2d);

    } // Fim do loop colunas

} // Fim do loop linhas
```

Se o tiro inimigo estiver inativo (ainda não foi disparado, ou colidiu com o jogador ou a linha de base), para o primeiro tiro, verificamos se a nave de onde sairá o disparo está à esquerda do tanque. Para o segundo, verificamos se houve uma colisão vertical, e para o terceiro, se a nave está à direita do tanque.

Se o `chefe` não estiver ativo e o número aleatório de 0 à 499 coincidir com o número de inimigos destruídos, teremos uma nave inimiga a mais na tela.

```
if (moverInimigos && novaLinha) {
    dir *= -1;
    novaLinha = false;

} else if (moverInimigos && colideBordas) {
    novaLinha = true;
}

moverInimigos = false;
...
```

Por fim, verificamos se houve movimentação com nova linha, ou movimentação e colisão. Para movimentação com nova linha, invertemos a direção (`dir` com valor positivo vai da esquerda para direita e, com valor negativo, da direita para esquerda) e desativamos a nova linha. Caso a segunda verificação seja verdadeira, a próxima movimentação será para uma nova linha. O método que adiciona tiros dos invasores é responsável por posicionar o tiro abaixo e no centro da nave que disparou.

```
...
public void addTiroInimigo(Elemento inimigo, Elemento tiro) {
    tiro.setAtivo(true);
    tiro.setPx(inimigo.getPx() +
               inimigo.getLargura() / 2 - tiro.getLargura() / 2);
    tiro.setPy(inimigo.getPy() + inimigo.getAltura());
}
...
```

Vale lembrar que, para os disparos começarem das linhas inferiores, o loop é feito percorrendo as linhas (o array dentro do array) de forma decrescente. No nosso código, o disparo só é efetuado quando o inimigo se movimenta, mas essa escolha, bem como outras (espaçamento, velocidade, tamanho da tela, elementos do jogo etc.) são abordagens que você deve escolher, deixando o modo de jogo ao seu gosto.

```
...
// Pressionou espaco, adiciona tiro
if (controleTecla[4] && !tiroTanque.isAtivo()) {
    tiroTanque.setPx(tanque.getPx() +
    tanque.getLargura() / 2 - tiroTanque.getLargura() / 2);

    tiroTanque.setPy(tanque.getPy() - tiroTanque.getAltura());
    tiroTanque.setAtivo(true);
}
...

if (chefe.isAtivo()) {
    chefe.incPx(tanque.getVel() - 1);

    if (!tiroChefe.isAtivo() && Util.colideX(chefe, tanque)) {
        addTiroInimigo(chefe, tiroChefe);
    }

    if (chefe.getPx() >canvas.getWidth()) {
        chefe.setAtivo(false);
    }
}
```

Os vários disparos desse jogo sempre partem do centro da nave ou do tanque, então, sempre estamos centralizando alguma coisa usando um código semelhante a este:

```
el.getPx() + el.getLargura() / 2 - tiro.getLargura() / 2
```

Em outras palavras, posição X do elemento origem mais metade do tamanho do elemento origem menos a metade do tamanho do elemento que queremos centralizar. Ou seja, alinhamos o centro da nave com o centro do tiro.

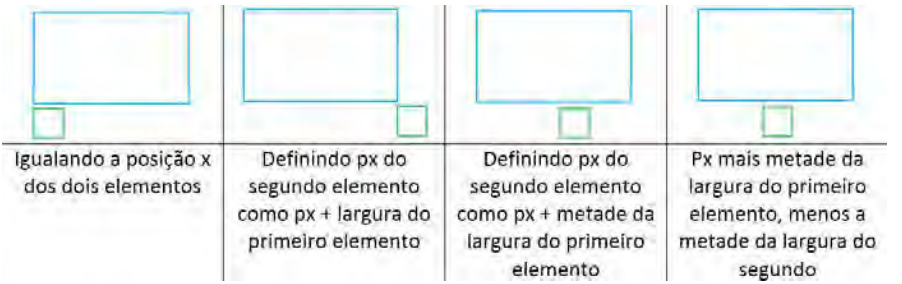

Fig. 2.3: A lógica do centralizar

Centralizar na horizontal é equivalente a substituir Px por Py e largura por altura. Essa é uma ótima função para ir para nossa classe `Util`, mas deixaremos por sua conta.

2.3 O disco chefe voador

O chefe do jogo (ou quase isso) é uma instância do objeto `Invader` que aparece de forma randômica, sendo que, no jogo original, a aparição é temporizada. Ele cruza a tela na horizontal e toda vez que avista o jogador (colide no eixo X), ele dispara um tiro mais longo e mais rápido que dos invasores comuns. Destacando o chefe e o tiro do chefe:

```
...
chefe = new Invader(Invader.Tipos.CHEFE);
```

```
tiroChefe = new Tiro(true);
tiroChefe.setVel(20);
tiroChefe.setAltura(15)
...

if (chefe.isAtivo()) {
    chefe.incPx(tanque.getVel() - 1);

    if (!tiroChefe.isAtivo() && Util.colideX(chefe, tanque)) {
        addTiroInimigo(chefe, tiroChefe);
    }

    if (chefe.getPx() > tela.getWidth()) {
        chefe.setAtivo(false);
    }
}
...
if (tiroChefe.isAtivo()) {
    tiroChefe.incPy(tiroChefe.getVel());

    if (Util.colide(tiroChefe, tanque)) {
        vidas--;
        tiroChefe.setAtivo(false);

    } else if (tiroChefe.getPy() > tela.getHeight() -
                        linhaBase - tiroChefe.getAltura()) {
        tiroChefe.setAtivo(false);
    } else
        tiroChefe.desenha(g2d);
}
...
```

Quando o `tiroChefe` está ativo, ele desce na vertical e, caso colida com o tanque, o jogador perde uma vida e o disparo é inativado. Se o tiro chegar à linha de base, também é inativado; caso contrário, ele é desenhado na tela.

2.4 Os invasores marcham

Aqui, como no jogo original, são três tipos diferentes de invasores, sendo que os maiores dão menos pontos que os menores. Além disso, eles têm dois estados diferentes, fazendo o desenho variar conforme marcham pela tela. Esse controle é feito na classe `Invader`, usando um `enum` para os tipos, e um `boolean` para variar o desenho.

```
package br.com.mvbos.lgj;

import java.awt.Color;
import java.awt.Graphics2D;
import br.com.mvbos.lgj.base.Elemento;

public class Invader extends Elemento {

    enum Tipos {
        PEQUENO, MEDIO, GRANDE, CHEFE
    }

    private Tipos tipo;
    private boolean aberto;

    public Invader(Tipos t) {
        this.tipo = t;

        setLargura(20);
        setAltura(20);
    }

    @Override
    public void atualiza() {
        aberto = !aberto;
    }
```

Toda vez que o método `atualiza` é chamado, o valor da variável `aberto` é invertido. A lógica do método `desenha` dá mais trabalho. Cada `tipo` de invasor tem seu próprio desenho, e o desenho muda dependendo do valor da variável `aberto`.

Já vimos como desenhar no primeiro capítulo, então o código foi omitido por enquanto, mas recomendo que pratique com seus próprios desenhos.

```
@Override
public void desenha(Graphics2D g) {

    if (!isAtivo())
        return;

    int larg = getLargura();

    if (tipo == Tipos.PEQUENO) {
        larg = larg - 2;
        g.setColor(Color.BLUE);

        if (aberto)
            // Desenha um círculo azul com quadrados ao redor
         else
            // Desenha um quadrado azul

    } else if (tipo == Tipos.MEDIO) {
        g.setColor(Color.ORANGE);
        if (aberto)
           // Desenha um quadrado vazio bordas na cor laranja
        else
           // Desenha um quadrado preenchido na cor laranja

    } else if (tipo == Tipos.GRANDE) {
        larg = larg + 4;
        if (aberto)
           // Desenha um retângulo em pé na cor cinza escuro
         else
           // Desenha um retângulo deitado na cor cinza

    } else {
        // Tenta desenhar algo parecido com um disco
        //                voador com luzes piscantes
        ...
        if (aberto) {
```

```
            // Três quadrados brancos
        }
    }
}
```

Além de utilizarmos o `enum Tipos` para determinar o formato do desenho, usamos para definir os pontos do jogador. Sendo que, se ele não for pequeno, médio ou grande, será o chefe, dando maior pontuação.

```
public int getPremio() {
        switch (tipo) {
        case PEQUENO:
            return 300;
        case MEDIO:
            return 200;
        case GRANDE:
            return 100;
        default:
            return 1000;
        }
    }
}
```

As classes `Tiro`, `Tanque` e `Texto`, que também são filhas da classe `Elemento`, são parecidas com a classe `Invader`, embora mais simples. O que define se o elemento `Tiro` será usado pelo tanque ou pelos inimigos é a propriedade `inimigo`.

```
package br.com.mvbos.lgj;

import java.awt.Color;
import java.awt.Graphics2D;
import br.com.mvbos.lgj.base.Elemento;

public class Tiro extends Elemento {

    private boolean inimigo;

    public Tiro() {
```

```
        setLargura(5);
        setAltura(5);
    }

    public Tiro(boolean inimigo) {
        this();
        this.inimigo = inimigo;
    }

    @Override
    public void atualiza() {
    }

    @Override
    public void desenha(Graphics2D g) {
        if (!isAtivo())
            return;

        g.setColor(inimigo ? Color.RED : Color.WHITE);
        g.fillRect(getPx(), getPy(), getLargura(), getAltura());
    }
}
```

Então, se o valor for verdadeiro, sabemos que o tiro é do inimigo e ele será desenhado na cor vermelha, caso contrário, na cor branca. Para que nosso `Tanque` tenha um formato de tanque (ou quase isso), fizemos três desenhos.

```
package br.com.mvbos.lgj;

import java.awt.Color;
import java.awt.Graphics2D;
import br.com.mvbos.lgj.base.Elemento;

public class Tanque extends Elemento {

    private final int cano = 8;
    private final int escotilha = 10;

    public Tanque() {
```

```
        setLargura(30);
        setAltura(15);
    }

    @Override
    public void atualiza() {
    }

    @Override
    public void desenha(Graphics2D g) {
        g.setColor(Color.GREEN);
        g.fillRect(getPx() + getLargura() / 2 - cano / 2,
              getPy() - cano, cano, cano);

        g.fillRect(getPx(), getPy(), getLargura(), getAltura());

        g.setColor(Color.YELLOW);
        g.fillOval(getPx() + getLargura() / 2 - escotilha / 2,
              getPy() + getAltura() / 2 - escotilha / 2,
              escotilha, escotilha);
    }
}
```

O primeiro para representar o cano de disparo, o segundo a carcaça e o terceiro a escotilha, esse último um círculo preenchido na cor amarela. Vale ressaltar que utilizamos somente uma instância da classe `Texto`, por isso, nela não armazenamos informações sobre o que será escrito, e sim com qual fonte, cor e tamanho será escrito. Isto porque todos os textos do jogo terão a mesma aparência.

```
package br.com.mvbos.lgj.base;

import java.awt.Font;
import java.awt.Graphics2D;

public class Texto extends Elemento {

    private Font fonte;
```

```
    public Texto() {
        fonte = new Font("Tahoma", Font.PLAIN, 16);
    }

    public Texto(Font fonte) {
        this.fonte = fonte;
    }

    public void desenha(Graphics2D g, String texto) {
        desenha(g, texto, getPx(), getPy());
    }

    public void desenha(Graphics2D g, String texto,
                        int px, int py) {
        if (getCor() != null)
            g.setColor(getCor());

        g.setFont(fonte);
        g.drawString(texto, px, py);
    }

    public Font getFonte() {
        return fonte;
    }

    public void setFonte(Font fonte) {
        this.fonte = fonte;
    }

}
```

Esse é realmente nosso primeiro jogo e alguns detalhes tiveram de ficar de fora para que pudéssemos codificar uma versão simples de ser criada, mas funcional. Conforme veremos ao longo do livro, mesmo um jogo simples de ser jogado não necessariamente quer dizer simples para ser codificado.

2.5 Codificando o jogo

Depois de tudo isso, o melhor para tirar qualquer dúvida é pegar uma versão e jogar, fazendo as modificações que quiser e ficando à vontade para compartilhar conosco em: https://github.com/logicadojogo/fontes/tree/master/Cap02.

Na seção a seguir, comentamos o que achamos que seria importante para o jogo e acabou não cabendo no capítulo. E na seção 2.5, deixamos um desafio para você, leitor. Para as duas seções, as respostas ou estão distribuídas em outros capítulos, ou serão vistas com maior profundidade no capítulo 10.

Não fizemos aqui

- Barreiras parecidas com as do jogo original.

Melhore você mesmo

- Impedir o jogador de sair da tela;
- Encerrar o jogo quando o tanque perder todas as vidas.

2.6 Resumo

Criar jogos é um exercício muito criativo e, se você conseguir imaginar formas de dar mais criatividade ao jogo, por favor, o faça e compartilhe. Nosso jogo já começa direto na ação e sabemos que não é bem assim que um bom jogo deve começar. Se ao menos ele tivesse *pause* (pausa), mas nem isso. Mas não se preocupe, já que esses são dois recursos que abordaremos no próximo capítulo.

Capítulo 3

Meu nome é Pong, Ping Pong

Na verdade, o nome é somente Pong, o primogênito da Atari Inc., projetado por Al Alcorn e lançado em 1972. Foi um grande sucesso.

Por ser um jogo simples de ser programado, vamos acrescentar mais funcionalidades e aumentar nossa biblioteca de código.

Neste capítulo, abordaremos com base em aspectos do jogo original:

- Interação com teclado e mouse;
- Separar o jogo em cenários;
- Menu simples;
- Movimentação diagonal;
- Configuração do modo de jogo.

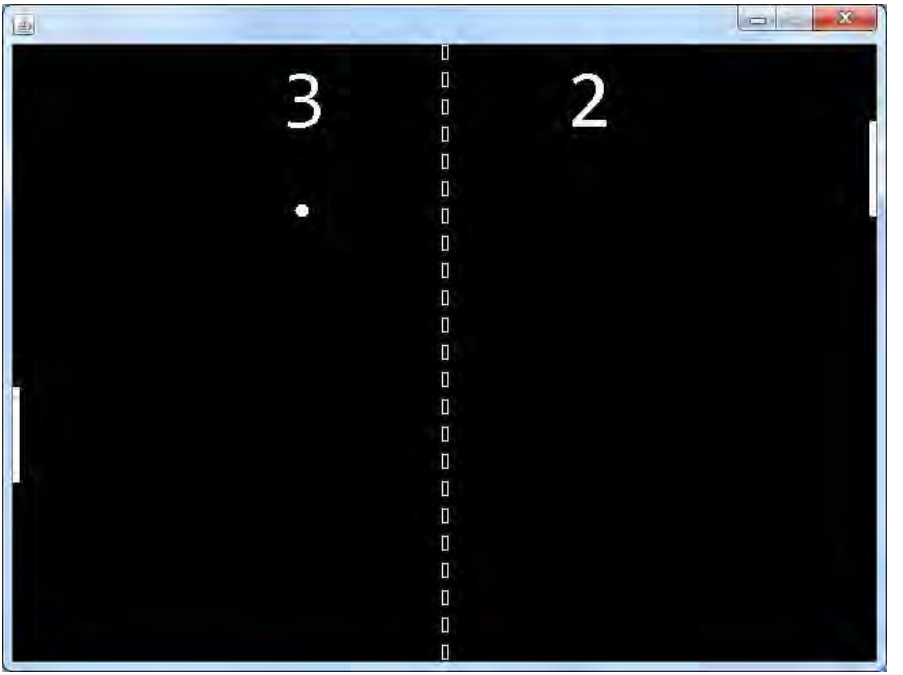

Fig. 3.1: Jogo modo normal

3.1 Separando a tela do cenário

Em nosso jogo anterior, a mesma classe responsável por criar a janela (`JFrame`), a tela (`JPanel`) e capturar a entrada do jogador também controlava o fluxo do jogo. Claramente podemos separar essas tarefas em duas classes distintas: uma que converse diretamente com o sistema operacional, e outra que fique responsável pelo do jogo. Além de termos um código mais enxuto e reaproveitável, fica mais fácil migrar o jogo para outras plataformas.

É ai que entra nossa classe `CenarioPadrao` para incrementar nossa caixa de ferramentas.

```
package br.com.mvbos.lgj.base;

import java.awt.Graphics2D;
```

```
public abstract class CenarioPadrao {

    protected int altura, largura;

    public CenarioPadrao(int largura, int altura) {
        this.altura = altura;
        this.largura = largura;
    }

    public abstract void carregar();

    public abstract void descarregar();

    public abstract void atualizar();

    public abstract void desenhar(Graphics2D g);
}
```

Assim, podemos dividir nossos jogos em diversos cenários, que não necessariamente serão fases do jogo, como por exemplo, a tela de introdução ou um menu de opções.

Neste jogo, temos duas classes que extendem da classe `CenarioPadrao`, `InicioCenario` e `JogoCenario`. A primeira é onde o jogador escolherá o modo de jogo (normal ou em casa) e a velocidade (normal, rápido e lento). A segunda é o jogo propriamente dito.

Nossa classe `Jogo` trabalhando com cenários ficou assim:

```
...
private CenarioPadrao cenario;
...
public void iniciarJogo() {
    long prxAtualizacao = 0;

    while (true) {
        if (System.currentTimeMillis() >= prxAtualizacao) {
            g2d.setColor(Color.BLACK);
            g2d.fillRect(0, 0, JANELA_LARGURA, JANELA_ALTURA);
```

```
            ...

            if (cenario == null) {
                g2d.setColor(Color.WHITE);
                g2d.drawString("Carregando...", 20, 20);
            } else {
                cenario.atualizar();
                cenario.desenhar(g2d);
            }

            tela.repaint();
            prxAtualizacao = System.currentTimeMillis() + FPS;
        }
    }
}
```

A troca entre os cenários funciona da seguinte forma: se o jogador estiver na tela de introdução e pressionar a tecla `enter` ou espaço, trocamos para o cenário do jogo, mas se estiver jogando, o jogo será pausado. Caso pressione `ESC` durante o jogo, voltamos para o cenário de introdução.

```
...
public enum Tecla {
    CIMA, BAIXO, ESQUERDA, DIREITA, BA, BB
}

public static boolean[] controleTecla =
            new boolean[Tecla.values().length];
...
public static boolean pausado;
...
if (controleTecla[Tecla.BA.ordinal()]) {
    // Pressionou espaço ou enter
    if (cenario instanceof InicioCenario) {
        cenario.descarregar();
        cenario = new JogoCenario(tela.getWidth(),
                        tela.getHeight());
        cenario.carregar();
```

```
    } else if (cenario instanceof JogoCenario) {
        Jogo.pausado = !Jogo.pausado;
    }

    liberaTeclas();

} else if (controleTecla[Tecla.BB.ordinal()]) {
    // Pressionou ESC
    if (cenario instanceof JogoCenario) {
        cenario.descarregar();
        cenario = new InicioCenario(tela.getWidth(),
                        tela.getHeight());
        cenario.carregar();
    }

    liberaTeclas();
}
```

Utilizamos o operador `instanceof` para sabermos em qual cenário estamos e, antes de ocorrer a troca, chamamos o método `descarregar`, seja para liberar memória ou para salvar algum estado do jogo.

Por comodidade, passamos a largura e altura da tela no construtor da classe. Mesmo que nossas telas não sejam redimensionadas, utilizamos muitas vezes o valor da largura e altura da tela em nossos cenários.

O método `carregar`, além de carregar o jogo, é útil para os casos em que querermos apenas reiniciá-lo, reposicionando todos os elementos no ponto de origem, o que seria mais rápido que criar uma nova instância.

> **Dica**
>
> Menus devem ser intuitivos e fáceis de usar. Logo, evite grandes e complexos menus, e não tenha receio de utilizar ícones e legendas.

Como você deve ter reparado, não estamos mais usando números para as teclas pressionadas, pois com o `enum Tecla` fica mais fácil saber qual tecla foi pressionada. Utilizamos `BA` e `BB` (Botão A e B) em vez do nome da tecla para ser independente de vínculos, facilitando mudanças.

Em nossos cenários, só precisamos saber se `BA` ou `BB` foram pressionados, sem precisarmos saber qual tecla está ligada a estes botões. Usamos o método `liberaTeclas` quando precisamos garantir que não executaremos a mesma ação mais de uma vez.

```
public static void liberaTeclas() {
    for (int i = 0; i < controleTecla.length; i++) {
        controleTecla[i] = false;
    }
}
```

Ele faz o oposto do método `setaTecla`, definindo todos os valores de `controleTecla` como falso. Por fim, nosso jogo tem dois jogadores, um controlado pelo teclado e outro pelo mouse, mas, diferente do que já vimos no exemplo `InterativoMouse` no capítulo 1, aqui ignoramos os cliques e pegamos apenas a posição do mouse no eixo Y.

```
...
public static int mouseY;
...
tela.addMouseMotionListener(new MouseMotionListener() {

    @Override
    public void mouseMoved(MouseEvent e) {
        mouseY = e.getY();
    }

    @Override
    public void mouseDragged(MouseEvent e) {
    }
});
...
```

Usamos `mouseY` para armazenar a posição do mouse, que é atualizada toda vez que a `tela` detectar sua movimentação.

Nossa classe `InicioCenario` terá esse visual:

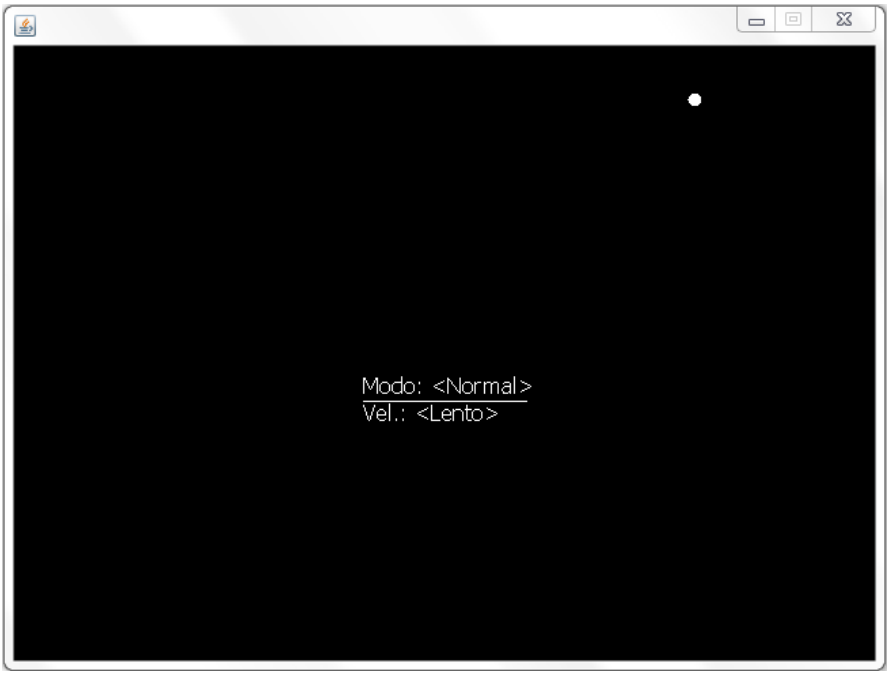

Fig. 3.2: Introdução do jogo

Temos dois menus: o primeiro com duas opções e o segundo com três. Para enfeitar o cenário introdutório, inserimos a bola do jogo.

```
...
private Bola bola;
private Menu menuModo;
private Menu menuVeloc;

@Override
public void carregar() {
    bola = new Bola();

    menuModo = new Menu("Modo");
    menuModo.addOpcoes("Normal", "Em casa");

    menuVeloc = new Menu("Vel.");
```

```
        menuVeloc.addOpcoes("Normal", "Rápido", "Lento");

        Util.centraliza(bola, largura, altura);
        Util.centraliza(menuModo, largura, altura);
        Util.centraliza(menuVeloc, largura, altura);

        menuModo.setPy(menuModo.getPy() + 20);
        menuVeloc.setPy(menuModo.getPy() + menuModo.getAltura());

        bola.setAtivo(true);
        menuModo.setSelecionado(true);
        menuModo.setAtivo(true);
        menuVeloc.setAtivo(true);
    }
```

Não se preocupe com o objeto `Bola` agora, veremos com mais detalhes quando falarmos do jogo em si, afinal, este é praticamente o astro do Pong. Criamos dois objetos `Menu`, que veremos em breve, e adicionamos as respectivas opções para cada um.

Depois de centralizarmos os três objetos, apenas distanciamos os menus, de forma que o `menuModo` não colida com a bola e o `menuVeloc` não colida com o `menuModo`. O método `descarregar` tem uma função importante, que é passar para o jogo as opções escolhidas.

```
    @Override
    public void descarregar() {
        Jogo.velocidade = bola.getVel();
        Jogo.modoNormal = menuModo.getOpcaoId() == 0;
    }
```

Vale notar que a bola que fica quicando na tela de introdução não serve somente de enfeite. Ao alterar a opção de velocidade, refletimos na bola que se move na velocidade escolhida, dando ao jogador uma indicação de qual será a velocidade dela dentro do jogo, por isso `Jogo.velocidade` recebe o valor de `bola.getVel()`.

As mudanças no menu são verificadas dentro do método `atualizar`:

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

```
    if (Jogo.controleTecla[Jogo.Tecla.CIMA.ordinal()] ||
        Jogo.controleTecla[Jogo.Tecla.BAIXO.ordinal()]) {
        if (menuModo.isSelecionado()) {
            menuModo.setSelecionado(false);
            menuVeloc.setSelecionado(true);

        } else {
            menuModo.setSelecionado(true);
            menuVeloc.setSelecionado(false);
        }
```

Quando o jogador pressiona para cima ou para baixo, alternamos entre os menus, e o menu que estiver selecionado será desenhado com sublinhado. Agora, quando ele pressiona para esquerda ou direita, o método `trocaOpcao` da classe `Menu` faz a mudança entre as opção definidas, indo para a esquerda (se o valor passado for verdadeiro) ou para a direita (se for falso), claro que isso somente se o menu em questão estiver selecionado.

```
} else if (Jogo.controleTecla[Jogo.Tecla.ESQUERDA.ordinal()]
       || Jogo.controleTecla[Jogo.Tecla.DIREITA.ordinal()]) {

        boolean esqueda =
            Jogo.controleTecla[Jogo.Tecla.ESQUERDA.ordinal()];

        menuModo.trocaOpcao(esqueda);
        menuVeloc.trocaOpcao(esqueda);

        if (menuVeloc.getOpcaoId() == 0) {
            bola.setVel(Bola.VEL_INICIAL);

        } else if (menuVeloc.getOpcaoId() == 1) {
            bola.setVel(Bola.VEL_INICIAL * 2);

        } else {
            bola.setVel(Bola.VEL_INICIAL / 2);
        }
    }

    Jogo.liberaTeclas();
```

```
    // Controle da bola
    ...
}
```

Dependendo da opção passada para o `menuVeloc`, atualizamos a velocidade da bola, sendo que a opção "Rápido" é o dobro da velocidade normal, e "Lento" é metade dessa velocidade. A classe `Menu` encontra-se no pacote base e faz parte da nossa caixa de ferramentas, servindo para outros jogos.

```
public class Menu extends Texto {

    private short idx;
    private String rotulo;
    private String[] opcoes;
    private boolean selecionado;

    public Menu(String rotulo) {
        super();
        this.rotulo = rotulo;
        setLargura(120);
        setAltura(20);
    }

    public void addOpcoes(String... opcao) {
        opcoes = opcao;
    }
```

Por trabalhar principalmente com escrita, nossa classe `Menu` herda de `Texto`, e não diretamente da nossa classe `Elemento`. No método construtor, definimos uma largura e altura padrão e, no método `desenha`, concatenamos o rótulo do menu e a última opção selecionada entre o sinal de `<` e `>`. Caso esteja selecionado, desenhamos uma linha abaixo do texto.

```
@Override
    public void desenha(Graphics2D g) {
        if (opcoes == null)
            return;
```

```
        g.setColor(getCor());
        super.desenha(g, getRotulo() + ": <" + opcoes[idx] + ">",
                getPx(), getPy() + getAltura());

        if (selecionado)
            g.drawLine(getPx(), getPy() + getAltura() + 5,
              getPx() + getLargura(), getPy() + getAltura() + 5);
    }

...

    public void trocaOpcao(boolean esquerda) {
        if (!isSelecionado() || !isAtivo())
            return;

        idx += esquerda ? -1 : 1;

        if (idx < 0)
            idx = (short) (opcoes.length - 1);
        else if (idx == opcoes.length)
            idx = 0;
    }
}
```

O método para navegar entre as opções (`trocaOpcao`) primeiro verifica se o menu está selecionado e se é um elemento ativo, para então navegar para a esquerda. Decrementamos `idx` e incrementamos para ir para a direita.

Se, por acaso, o `idx` for menor do que zero (que é a primeira opção), posicionamos o menu na última opção; caso ele passe da última opção, posicionamos na primeira.

Utilizamos um tipo `short` para o `idx`, que ocupa menos espaço em memória que um tipo `int`, isso fará diferença se estiver desenvolvendo jogos para plataformas com pouca memória.

Dica

Estamos concatenando os textos rótulo e opção do menu, mas o Java nos fornece uma forma mais elegante com o método `String.format`. Assim, podemos substituir o código anterior pela chamada: `String.format("%s: <%s>", getRotulo(), opcoes[idx])`, onde cada `%s` será substituído pelos parâmetros seguintes na mesma ordem em que aparecem.

Usamos `%s` para `String` (texto), `%d` para inteiros e `%f` para ponto flutuante.

3.2 Jogando o código

A maior parte da lógica do jogo está ligada à bola (nossa classe `Bola`), que tem de quicar e ser rebatida em diferentes velocidades e ângulos. Para isso, usamos dois inteiros para controlar a direção da bola, e dois flutuantes para controlar a velocidade, ambos para os eixos X e Y, respectivamente.

Com isso, podemos ter uma bola indo rapidamente para a esquerda, mas subindo lentamente. Ou seja, com direções e velocidades diferentes para cada eixo, deixando o movimento mais natural.

```
public class Bola extends Elemento {

    public static final int VEL_INICIAL = 3;
    private int dirX = -1;
    private int dirY = -1;
    private float velX;
    private float velY;

    public Bola() {
        velX = velY = VEL_INICIAL;
        setAltura(10);
        setLargura(10);
        setCor(Color.WHITE);
    }
```

```
...

    @Override
    public void setVel(int vel) {
        velX = velY = vel;
    }

    @Override
    public int getVel() {
        return (int) velX;
    }

    public void incPx() {
        incPx((int) velX * dirX);
    }

    public void incPy() {
        incPy((int) velY * dirY);
    }

    public void inverteX() {
        dirX *= -1;
    }

    public void inverteY() {
        dirY *= -1;
    }
}
```

Já trabalhamos com direção no jogo anterior, aqui estamos apenas utilizando a mesma lógica nos dois eixos com `dirX` e `dirY`. Iniciamos nossa `bola` com a velocidade, largura, altura e cor padrão.

Observe que o método `setVel` atualiza tanto `velX` quanto `velY`. Não usamos o atributo `vel` da classe `Elemento` para evidenciar melhor de qual eixo estamos mudando a velocidade.

E agora, sem mais delongas, vamos à nossa classe `JogoCenario`, onde dois jogadores deslizam na vertical rebatendo um ponto branco que vai ficando cada vez mais rápido, até que um deles erre e o outro faça ponto. Co-

meçamos declarando as seguintes variáveis:

```
...
private float inc = 0.5f;
private Ponto pontoA, pontoB;
private Bola bola;
private Elemento esquerda;
private Elemento direita;
private boolean reiniciarJogada;
private final Texto textoPausa = new Texto(Ponto.fonte);

// Modo em casa
private int idx;
private Bola[] bolaArr = new Bola[0];
private Random rand;
...
```

Utilizamos `inc` para incrementar a velocidade da bola quando ela é rebatida. Usamos o tipo flutuante para um aumento mais gradual. Como pode imaginar, `pontoA` e `pontoB` são usados para exibir a pontuação de cada jogador, que são representados por: `esquerda` e `direita`.

Durante a pausa do jogo, será exibido o texto `PAUSA`, com a mesma fonte usada para exibir a pontuação; usamos `textoPausa` para isso. Esse jogo tem dois modos: um mais parecido com o original, e outro no qual, a cada rebatida, uma nova bola surge no centro da tela, com tamanho e velocidades variadas, batizado de *Em casa*. Para isso, são usadas as três últimas variáveis.

```
...
public JogoCenario(int largura, int altura) {
    super(largura, altura);
    bola = new Bola();
    esquerda = new Elemento();
    direita = new Elemento();
    pontoA = new Ponto();
    pontoB = new Ponto();
}
...
```

Iniciamos os principais elementos do jogo no construtor da classe, deixando para o método `carregar` o trabalho de configurá-los.

```
...
@Override
public void carregar() {
    bola.setVel(Jogo.velocidade);

    pontoA.setPx(largura / 2 - 120);
    pontoA.setPy(Ponto.TAMANHO_FONTE);

    pontoB.setPx(largura / 2 + 120 - Ponto.TAMANHO_FONTE / 2);
    pontoB.setPy(Ponto.TAMANHO_FONTE);

    esquerda.setVel(5);
    esquerda.setAltura(70);
    esquerda.setLargura(5);
    esquerda.setCor(Color.WHITE);

    direita.setVel(5);
    direita.setAltura(70);
    direita.setLargura(5);
    direita.setCor(Color.WHITE);
    direita.setPx(largura - direita.getLargura());

    Util.centraliza(bola, largura, altura);
    Util.centraliza(direita, 0, altura);
    Util.centraliza(esquerda, 0, altura);

    bola.setAtivo(true);
    direita.setAtivo(true);
    esquerda.setAtivo(true);
...
```

A bola inicialmente e toda vez que a jogada é reiniciada recebe a velocidade definida na introdução do jogo. Os dois objetos da classe `Ponto`, que estendem a classe `Texto`, são posicionados próximos da linha central de forma bem simples, variando de acordo com o tamanho da fonte.

Vale ressaltar que, para posicionar corretamente a pontuação, além do

tamanho da fonte, teríamos de considerar a quantidade de caracteres e recalcular a posição conforme eles são alterados. Sem recalcularmos, conforme a pontuação do jogador da esquerda for aumentando, ela invadirá o campo do jogador da direita, isso acima dos mil pontos.

Ainda no método `carregar`, se o modo de jogo não for o modo normal, iniciaremos o `bolaArr` com tamanho de 30 e, então, criamos 30 objetos (bolas), variando direção, velocidade, largura e altura.

```
...
    if (!Jogo.modoNormal) {
        rand = new Random();
        bolaArr = new Bola[30];

        for (int i = 0; i < bolaArr.length; i++) {
            int v = rand.nextInt(3) + 1;

            bolaArr[i] = new Bola();
            bolaArr[i].setDirX(i % 2 == 0 ? -1 : 1);

            bolaArr[i].setVel(Bola.VEL_INICIAL * v);
            bolaArr[i].setAltura(bola.getAltura() * v);
            bolaArr[i].setLargura(bola.getLargura() * v);

            Util.centraliza(bolaArr[i], largura, altura);
        }
    }
}
```

O valor de `v` pode ser de 1 a 3. Assim, se o valor for 1, será uma bola padrão, mas se for 3, será três vezes mais rápida e maior.

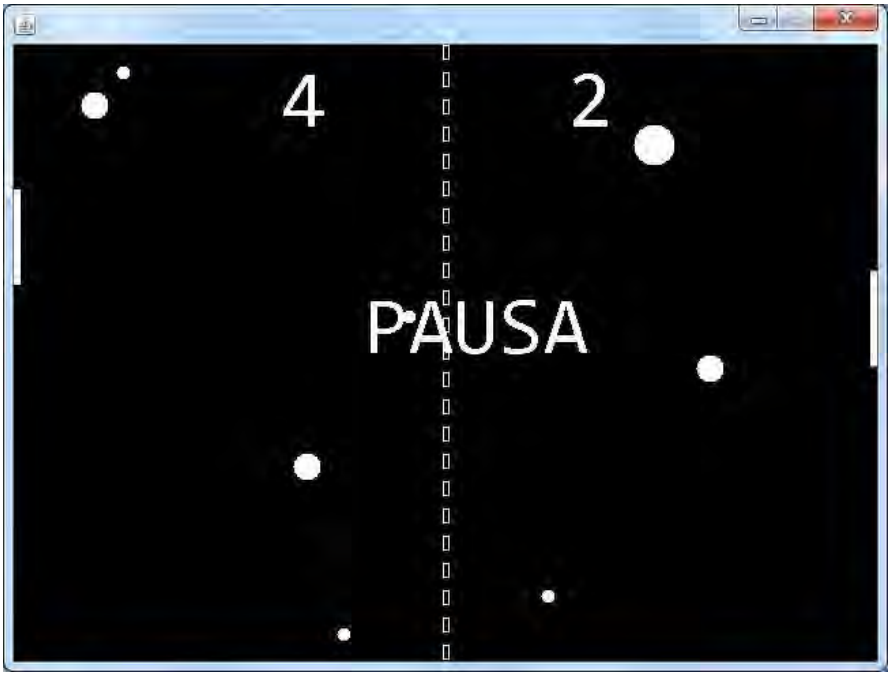

Fig. 3.3: Modo feito em casa com pausa

> **Dica**
>
> Se a velocidade da bola for superior à largura da raquete, a chance de ela não ser pega pelo nosso detector de colisão, atravessando o jogador, é grande.

O método `atualiza` é o juiz dessa partida:

```
@Override
public void atualizar() {

    if (Jogo.pausado)
        return;

    bola.incPx();
```

```
    bola.incPy();

    if (Jogo.controleTecla[Jogo.Tecla.CIMA.ordinal()]) {
        esquerda.incPy(esquerda.getVel() * -1);
    } else if (Jogo.controleTecla[Jogo.Tecla.BAIXO.ordinal()]) {
        esquerda.incPy(esquerda.getVel());
    }

    if (direita.getPy() + direita.getAltura() / 2 >
        Jogo.mouseY + direita.getVel())
        direita.incPy(direita.getVel() * -1);
    else if (direita.getPy() + direita.getAltura() / 2 <
        Jogo.mouseY - direita.getVel())
        direita.incPy(direita.getVel());

    validaPosicao(esquerda);
    validaPosicao(direita);

    if (reiniciarJogada) {
        reiniciarJogada = false;
        bola.inverteX();
        bola.setVel(Jogo.velocidade);
        Util.centraliza(bola, largura, altura);

    } else {
        reiniciarJogada = validaColisao(bola);
    }

...

}
```

Antes de qualquer atualização do cenário ser feita, verificamos se o jogo está pausado. Caso esteja, nada mais será atualizado até que valor de `Jogo.pausado` mude para falso.

A movimentação do jogador da direita usa a posição da raquete no eixo Y mais a metade da altura dela para posicioná-la centralizada em relação ao ponteiro do mouse. Na verificação, somamos ou subtraímos a velocidade da

raquete para aumentar o limite do posicionamento, evitando que o objeto fique trepidando na tela.

O método `validaPosicao` apenas não deixa que os objetos acabem fora da tela no eixo Y. Quando reiniciamos a jogada, invertemos a direção da bola, então, quem marcar ponto terá que defender. Também voltamos a velocidade e posição inicial dela. O método `validaColisao` também é responsável pela pontuação:

```
private boolean validaColisao(Bola b) {
    boolean saiu = false;

    if (Util.colide(esquerda, b)) {
        rebate(esquerda, b);

    } else if (Util.colide(direita, b)) {
        rebate(direita, b);

    } else if (b.getPx() < 0 ||
            b.getPx() + b.getLargura() > largura) {

        saiu = true;

        if (b.getPx() < 0)
            pontoB.add();
        else
            pontoA.add();

    } else if (b.getPy() <= 0 ||
            b.getPy() + b.getAltura() >= altura) {
        // Colisão no topo ou base da tela
        b.inverteY();
    }

    return saiu;
}
```

Se a bola colidir com alguma das raquetes, ela será rebatida, mas se ela agilmente escapar delas, conseguindo sair da tela pelas laterais, o jogador do

lado oposto de onde a bola saiu marca ponto. Por fim, verificamos se a bola colidiu com o topo ou a base da tela, então, invertemos sua direção no eixo Y.

O método para rebater a bola poderia simplesmente inverter o eixo X, mas ele é um pouco mais sofisticado que isso. Quando uma das raquetes colide com a bola, temos cinco tipos diferentes de rebatidas: para uma colisão acima do centro, no centro e abaixo do centro, e para colisão com as quinas da raquete.

Fig. 3.4: Pontos de colisão raquete

Para isso, dividimos a raquete em três partes (que junto com a quina superior e inferior somarão cinco variações), e verificamos em qual dessas partes a bola bateu:

```
public void rebate(Elemento raquete, Bola bola) {
    float vx = bola.getVelX();
    float vy = bola.getVelY();

    if (bola.getPy() <
              raquete.getPy() + raquete.getAltura() / 3) {

        bola.setDirY(-1);

        vx += inc;
        vy += inc;

        if (bola.getPy() < raquete.getPy()) {
            vy += inc;
        }

    } else if (bola.getPy() > raquete.getPy() +
```

```
            raquete.getAltura() - raquete.getAltura() / 3) {
        bola.setDirY(1);

        vx += inc;
        vy += inc;

        if (bola.getPy() + bola.getAltura() >
            raquete.getPy() + raquete.getAltura()) {
            vy += inc;
        }

    } else {
        vx += inc;
        vy = 1;
    }

    bola.inverteX();
    bola.incVel(vx, vy);

    if (bolaArr.length > 0) {
        if (idx < bolaArr.length) {
            bolaArr[idx++].setAtivo(true);

        } else {
            idx = 0;
        }
    }
}
```

Se a bola bater no terço de cima, ela é rebatida para cima (passamos -1 para `dirY`), e adicionamos velocidade em ambos os eixos. Se o eixo Y da bola for menor que o da raquete (quina), incrementamos um pouco mais a velocidade no eixo Y.

Se a colisão for no terço de baixo, faremos algo similar. Caso seja no terço central, incrementamos `velX` e deixamos `velY` com 1, fazendo a bola voltar quase em linha reta. Estamos usando um tipo flutuante para termos mais opções de velocidade.

Depois de invertemos a direção no eixo X (`bola.inverteX`) e passar-

mos as novas velocidades para a bola (`bola.incVel(vx, vy)`), verificamos se ativamos uma nova bola, sendo que `bolaArr.length` vai ser maior que zero dependendo do modo de jogo.

Não usamos o ângulo da bola como base de cálculo para as rebatidas como é feito no jogo original, resultando em menos variação. Mas não se preocupe, temos um jogo específico só para abordar ângulos e rotações (consegue imaginar qual é?). Então, você poderá deixar o jogo mais próximo do original se quiser.

O método para desenhar nossos elementos ficou assim:

```
@Override
public void desenhar(Graphics2D g) {
    // Desenha linha de fundo
    for (int i = 0; i < altura; i += 20) {
        g.setColor(Color.WHITE);
        g.drawRect(largura / 2 - 2, i, 4, 10);
    }

    pontoA.desenha(g);
    pontoB.desenha(g);

    // depurarColisao(esquerda, g);
    // depurarColisao(direita, g);

    bola.desenha(g);
    for (Bola b : bolaArr) {
        b.desenha(g);
    }

    esquerda.desenha(g);
    direita.desenha(g);

    if (Jogo.pausado)
        textoPausa.desenha(g, "PAUSA",
            largura / 2 - Ponto.TAMANHO_FONTE,
            altura / 2);

}
```

Nada que já não tenhamos visto antes. Você pode usar o método `depurarColisao` para visualizar a divisão na raquete. Por fim, mas não menos importante, nossa classe `Ponto`:

```
...
public class Ponto extends Texto {

    public static final int TAMANHO_FONTE = 60;
    public static final Font fonte =
        new Font("Consolas", Font.PLAIN, TAMANHO_FONTE);

    private short ponto;

    public Ponto() {
        super.setFonte(fonte);
    }

    public short getPonto() {
        return ponto;
    }

    public void setPonto(short ponto) {
        this.ponto = ponto;
    }

    public void add() {
        ponto++;
    }

    @Override
    public void desenha(Graphics2D g) {
        super.desenha(g, Short.toString(ponto),
                  getPx(), getPy());
    }

}
```

Utilizamos novamente um tipo `short`, dessa vez nos pontos, para poupar memória, já que será difícil um jogador passar da marca de 32.767 pontos.

Poupar memória é uma coisa a se ter em mente quando se programa jogos (ou qualquer sistema). Você pode pensar que o jogo é simples demais ou que os computadores de hoje são poderosos o suficiente para não ter de se preocupar com isso, mas sempre existe a possibilidade de portar seu jogo para um celular (e nem digo smartphone), ou um dispositivo com menos memória.

Mas então qual o motivo de não utilizarmos `short` em quase toda parte do nosso código? Para não termos de fazer conversão toda hora, já que a maioria dos métodos do Java que usamos em nossos jogos trabalha com inteiros ou flutuantes.

3.3 Codificando o jogo

Se você pensou em Asteroids como o jogo em que falaremos de ângulos (capítulo 8), acertou. Nossa versão do jogo sem precisar calcular ângulos está em: https://github.com/logicadojogo/fontes/tree/master/Cap03.

Lá você pode conferir com mais detalhes as diversas alterações que fizemos na classe `Jogo.java` para trabalharmos com cenários, além das novas classes: `Bola.java`, `InicioCenario.java`, `JogoCenario.java` e `Ponto.java`.

Temos dois jogos e, se você unir Space Invaders com Pong, poderá criar o jogo Breakout, idealizado por Nolan Bushnell e Steve Bristow, lançado pela Atari em 1976.

Não fizemos aqui

- Calcular o ângulo da bola na rebatida;
- Exibir as últimas pontuações no cenário de introdução do jogo.

Melhore você mesmo

- Que tal um modo de jogo seu?

3.4 Resumo

Temos dois cenários aqui: um para o jogador escolher o modo e velocidade do jogo, e outro em que o jogo acontece. Poderíamos separar esses dois cenários com um ou mais `ifs`, entretanto, isso deixaria nosso código bagunçado, dando trabalho para manutenção e detecção de erros. Além do mais, teríamos objetos que só precisariam existir no cenário de introdução misturados com objetos do cenário do jogo.

Outro ponto seria a criação de novos cenários, por exemplo, um jogo como o Sonic 2, no qual temos a tela de introdução, tela de opções, tela do jogo e a tela de bônus que se comportam de maneiras diferentes.

Embora não estejamos criando nada parecido com os jogos do Sonic (e eu não saiba exatamente como ele foi projetado), separar em cenários (não me refiro a fases) em uma linguagem Orientada a Objeto é uma abordagem vantajosa.

Nossa classe `CenarioPadrao` é abstrata para que seja implementada por um cenário de verdade. E nossa classe principal (`Jogo`), ao tratar dos cenários, se preocupa basicamente em avisar ao cenário quando é hora de `atualizar` e `desenhar` seus elementos e fazer a troca de cenário, chamando os métodos `descarregar`, se já estiver com um cenário em execução, e `carregar`, caso não esteja. A pausa implementada é simplesmente parar de atualizar os elementos e continuar pintando a tela.

A utilização de constantes para controlar as teclas pressionadas beneficia a mudança de teclas/botões do jogo. Por exemplo, se em vez de usar **seta para cima** passássemos a utilizar **W**, a troca seria feita somente no método `setaTecla`.

No cenário inicial, ao pressionar para cima ou para baixo, é feita a troca de seleção entre os menus. Ao pressionar esquerda ou direita, o menu que estiver selecionado vai para a próxima opção.

O que vale destacar para o cenário do jogo (`JogoCenario`) são as modificações que fizemos para o modo *Em casa*. Declaramos variáveis, mas só criamos a instância caso não seja o *modo normal*, e fazemos isso quando o jogo é carregado, e não em tempo de execução.

Alocamos toda a memória que precisamos e gerenciamos os elementos ativando e desativando-os, sem que sejam criadas novas instâncias. A vanta-

gem desta abordagem é a performance, embora não seja uma diferença notável para este jogo. Mas fica como dica.

Capítulo 4

Jogo da cobrinha

Nem Blockade, desenvolvido pela Gremlin em 1976, nem Surround, desenvolvido pela Atari em 1977. Jogo da cobrinha era como eu chamava a versão que vinha com o QBasic (Nibbles).

Foi a primeira vez que tive contato com o código-fonte de um jogo e pude modificá-lo para criar outros níveis. Então, não foi casual escolher esse jogo para falar desse assunto; afinal, quem nunca jogou esse jogo em um celular antigo, na fila do banco ou mesmo no banheiro?

Veremos neste capítulo:

- Interação com teclado;
- Criação de níveis/fases/*levels*.

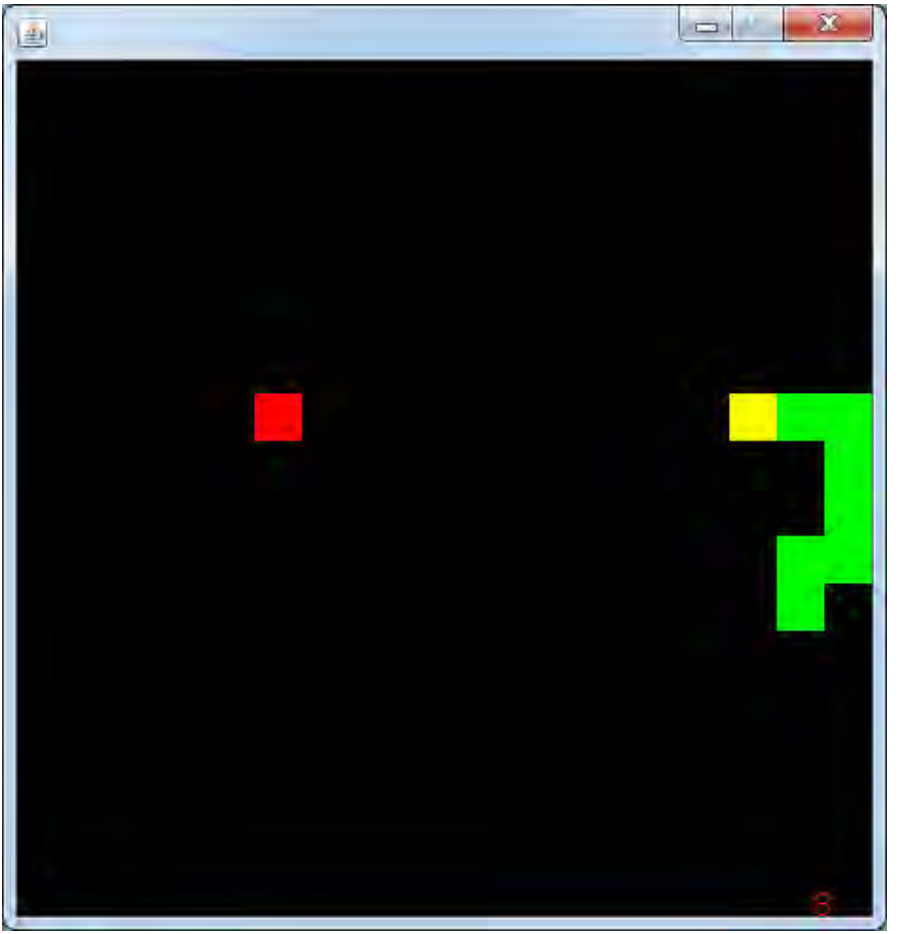

Fig. 4.1: Jogo da cobrinha

4.1 Níveis: aquilo que você faz para reaproveitar o cenário

No capítulo anterior, dividimos nosso código para termos uma classe lidando com o sistema operacional e outra para cuidar da lógica do nosso jogo, o cenário. Imaginando um jogo no estilo *Super Mario Bros. 3*, definitivamente não precisamos de um cenário novo para cada fase do jogo, já que muitas

das mecânicas, como correr, pular e comer cogumelos, serão reutilizadas em outras fases.

Faremos isso neste capítulo, usaremos nosso cenário de forma que ele tenha outros níveis, reaproveitando a mecânica do jogo.

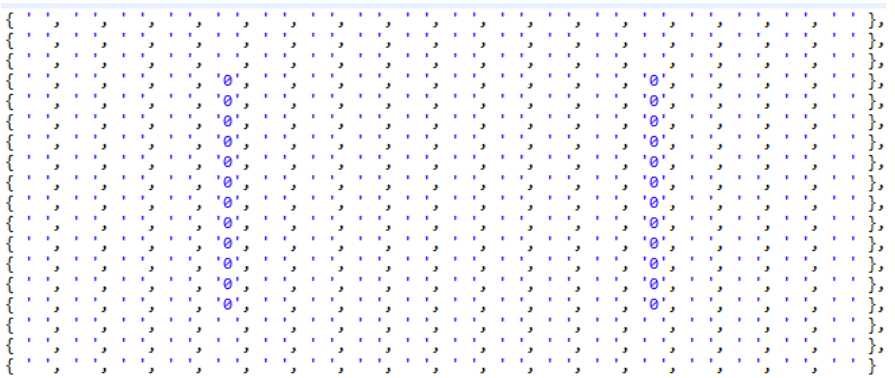

Fig. 4.2: Código da fase

Na figura 4.2, temos o código que representa a fase e, na figura 4.3, temos o resultado dessa fase no cenário. São apenas duas linhas verticais que formamos, utilizando zeros e espaço em branco para as áreas vazias.

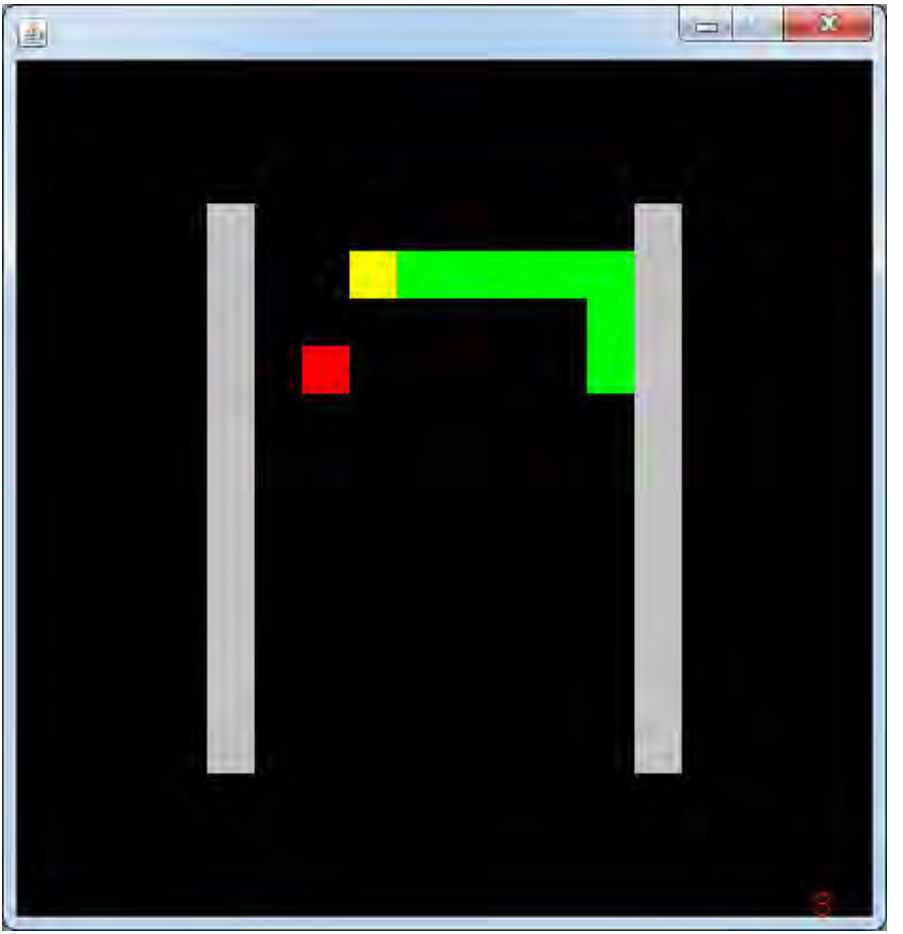

Fig. 4.3: Fase

Alguns jogos trabalham com níveis de forma intrínseca no código, outros nem tanto, e a maioria utiliza arquivos independentes. A maioria também usa ferramentas à parte para criação e edição de níveis (*Level Editor*), às vezes da própria empresa, outras, de terceiros. Ainda, algumas desenvolvedoras distribuem essas ferramentas para que o jogador possa expandir o universo do jogo.

Em nosso caso, separamos nosso código contendo os níveis na classe `Nivel`, e utilizamos um array tridimensional estático do tipo `char` (que dei-

xamos pré-formatado para facilitar a edição).

```java
public class Nivel {
    //18 por 18
    public static char[][][] niveis = {
            {
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '} },

            {
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '} },

            {
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', '0', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '0', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', '0', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '0', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', '0', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '0', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', '0', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '0', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', '0', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '0', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', '0', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '0', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', '0', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '0', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', '0', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '0', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', '0', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '0', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', '0', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '0', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', '0', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '0', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', '0', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '0', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '} } };
}
```

Fig. 4.4: Um jogo, três fases

Mas não se preocupe, continuamos trabalhando com array bidimensional (18 por 18):

...

```
char[][] nivelSelecionado = Nivel.niveis[Jogo.nivel];
nivel = new Elemento[nivelSelecionado.length * 2];

for (int linha = 0; linha < nivelSelecionado.length; linha++) {
    for (int coluna = 1; coluna < nivelSelecionado[0].length;
    coluna++) {
        if (nivelSelecionado[linha][coluna] != ' ') {

            Elemento e = new Elemento();
            e.setAtivo(true);
            e.setCor(Color.LIGHT_GRAY);

            e.setPx(_LARG * coluna);
            e.setPy(_LARG * linha);

            e.setAltura(_LARG);
            e.setLargura(_LARG);

            nivel[contadorNivel++] = e;
        }
    }
}
...
```

A variável `Jogo.nivel` recebe o valor do nível que selecionamos no cenário introdutório do jogo (similar ao jogo do capítulo anterior). Logo, obtemos um array bidimensional representando as linhas e colunas do cenário que devem ser preenchidas, neste caso, os valores diferentes de um espaço em branco.

Cada espaço preenchido será representado por um `Elemento` na cor cinza, tendo as mesmas dimensões dos outros objetos do jogo. Então, acabamos utilizando um array (desta vez, unidimensional), tendo como tamanho o dobro de linhas do nível selecionado (neste caso, 36), já que os níveis de exemplo que criamos não precisam de mais espaço que isso.

Observe que, devido ao fato de nossa serpente começar na linha 0 e coluna 0, ignoramos a coluna inicial, começando a `coluna` com o valor 1. Multiplicando `_LARG` pela `coluna`, temos a posição no eixo X, e `_LARG` pela

`linha`, a posição do elemento no eixo Y.

> **Dica**
>
> Use números ou letras diferentes para criar blocos de cores diferentes.

A variável `contadorNivel` é usada para percorrermos o array de acordo com o número de elementos, e não pelo comprimento, uma vez que o comprimento sempre será maior ou igual ao contador. Sabendo que nossos blocos não podem ocupar a tela inteira (pois a serpente não teria como se mover), uma forma melhor seria criar o array com o tamanho igual à quantidade de blocos definida nos níveis.

Uma forma não tão boa seria percorrer o array inteiro – que para um cenário maior, teria bem mais que 36 posições –, sem utilizar o contador. Ficamos no meio termo, ganhando pouca performance, que é melhor que nenhuma.

Tamanho da tela versus tamanho dos níveis

Nosso jogo roda em uma janela relativamente pequena, isso porque o tamanho da tela e a quantidade de linhas e colunas dos níveis que criamos estão ligados ao tamanho dos nossos elementos no jogo.

Por exemplo, as dimensões da nossa tela são 450px, e nossas fases têm 18 linhas e colunas. Dividindo 450 por 18, nós temos um espaçamento de 25 pixels para nossos elementos. Se aumentarmos a nossa tela, mantendo o mesmo número de linhas e colunas, nossos elementos que compõem os níveis do cenário terão de ocupar um espaço maior para serem posicionados corretamente. Isso quer dizer que não necessariamente teremos mais espaço.

Fixamos o tamanho em 18 linhas e colunas para facilitar a edição sem precisarmos de um editor de levels. Mesmo assim, já temos trabalho, e eu sei que você mal pode esperar pelo capítulo 6, onde trabalhamos com 31 linhas e 28 colunas.

Os elementos poderiam ter um tamanho menor ou maior que 25 pixels,

mas ocupamos o tamanho todo para não nos preocuparmos em centralizar o elemento para posicioná-lo corretamente na tela.

Nos capítulos anteriores, apenas nos preocupávamos com o tamanho da janela (`JFrame`), mas neste jogo, o tamanho da tela (`JPanel`) é o que importa, e ela não pode perder espaço para bordas. Assim, fizemos algumas modificações, e agora a criação da nossa tela ficou assim:

```
...
tela = new JPanel() {
    private static final long serialVersionUID = 1L;

    @Override
    public void paintComponent(Graphics g) {
        g.drawImage(buffer, 0, 0, null);
    }

    @Override
    public Dimension getPreferredSize() {
        return new Dimension(JANELA_LARGURA, JANELA_ALTURA);
    }

    @Override
    public Dimension getMinimumSize() {
        return getPreferredSize();
    }
};

getContentPane().add(tela);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setResizable(false);
pack();
...
```

Em vez de chamarmos o método `setSize(JANELA_LARGURA, JANELA_ALTURA)` do `JFrame`, sobrescrevemos os métodos `getPreferredSize()` e `getMinimumSize()` do `JPanel` e chamamos o método `pack()`, que basicamente diz para a janela que o espaço de que precisamos é de acordo com o tamanho dos elementos que estão nela,

garantindo que nossa tela terá o espaço que ela merece.

> **Dica**
>
> Para saber de quantas linhas e colunas você precisa baseado no tamanho da tela, divida o seu tamanho pelo tamanho que o elemento vai ocupar nela.
>
> Para uma tela de 650px de largura, em que os elementos ocuparão 25px de largura, por exemplo, precisamos de 26 colunas (650 / 25 = 26).

4.2 Jogando o código

Este vai ser o primeiro jogo em que realmente teremos início, meio e fim. O jogador vai jogar até perder, colidindo com o cenário ou ele mesmo; ou ganhar, comendo todas as frutas da tela.

Começamos pelas principais variáveis do jogo, que são:

```
enum Estado {
    JOGANDO, GANHOU, PERDEU
}

private static final int _LARG = 25;
private static final int RASTRO_INICIAL = 5;
...
private int contadorRastro = RASTRO_INICIAL;
```

Nosso `enum Estado` é usado para controlar o fluxo do jogo, que começa no estado jogando, no qual o jogador pode controlar a serpente; caso contrário, uma mensagem é exibida avisando se ele ganhou ou perdeu.

Para não parecer estranha, nossa serpente começa inicialmente com 5 rastros, e utilizamos um contador (`contadorRastro`) para controlar a adição de rastros no jogo.

```
private Elemento fruta;
private Elemento serpente;
private Elemento[] nivel;
```

```
private Elemento[] rastros;
...

// Frutas para finalizar o level
private int dificuldade = 10;
```

Temos apenas um objeto `fruta` no jogo, que é reposicionado ao colidir com o objeto `serpente`. Sendo que temos nenhum ou mais elementos que compõem o nível do cenário. A quantidade de rastros que o jogador precisa ter (frutas que ele deve comer) é controlada pela variável `dificuldade`.

Nosso método `carregar()`, sem as linhas de código para carregar o nível, que já vimos anteriormente, ficou assim:

```
...
// define direção inicial
dy = 1;
rastros = new Elemento[dificuldade + RASTRO_INICIAL];
fruta = new Elemento(0, 0, _LARG, _LARG);
fruta.setCor(Color.RED);
serpente = new Elemento(0, 0, _LARG, _LARG);
serpente.setAtivo(true);
serpente.setCor(Color.YELLOW);
serpente.setVel(Jogo.velocidade);

for (int i = 0; i < rastros.length; i++) {
    rastros[i] = new Elemento(serpente.getPx(), serpente.getPy(),
                 _LARG, _LARG);
    rastros[i].setCor(Color.GREEN);
    rastros[i].setAtivo(true);
}

...
```

A movimentação da serpente tem pontos importantes que ocorrem no método `atualizar()`. Primeiro, ela não se move continuamente, mas sim dentro de um intervalo (controlado pela variável `temporizador`), e este pode ser maior ou menor de acordo com a velocidade que o jogador escolher para o jogo.

Segundo, a quantidade de pixels utilizados na movimentação é referente à largura/altura padrão, que usamos para definir o tamanho de todos os elementos. Assim, o jogador tem maior controle nas manobras.

O terceiro ponto é que, toda vez que o primeiro bloco se move (objeto `serpente`), todo o corpo segue junto (objeto `rastros`), e ela está sempre indo para alguma direção da tela.

```
...
if (temporizador >= 20) {
    temporizador = 0;
    moveu = false;

    int x = serpente.getPx();
    int y = serpente.getPy();

    serpente.setPx(serpente.getPx() + _LARG * dx);
    serpente.setPy(serpente.getPy() + _LARG * dy);

...

    for (int i = 0; i < contadorRastro; i++) {
        Elemento rastro = rastros[i];
        int tx = rastro.getPx();
        int ty = rastro.getPy();

        rastro.setPx(x);
        rastro.setPy(y);

        x = tx;
        y = ty;
    }

} else {
    temporizador += serpente.getVel();
}
```

Para entender a lógica da movimentação da serpente, basta pensar em uma fila indiana, quando o primeiro elemento da fila se move, o segundo

ocupa a posição anterior do primeiro, o terceiro do segundo, e assim sucessivamente. Em nosso código, primeiramente, pegamos a posição atual do primeiro bloco (cabeça da serpente) e salvamos nas variáveis `x` e `y`. Depois, movemos a cabeça para a direção seguinte. Percorremos todo o rastro, fazendo o mesmo processo, guardando a posição atual (em `tx` e `ty`), depois movendo para a posição seguinte, até os rastros acabarem.

Ainda falando da movimentação da serpente, quando ela está se movendo na vertical, ela só pode virar na horizontal e, ao se mover na vertical, virar na horizontal. Assim, evitamos que o jogador dê "ré" e colida com o rastro.

```
if (!moveu) {
    if (dy != 0) {
        if (Jogo.controleTecla[Jogo.Tecla.ESQUERDA.ordinal()]) {
            dx = -1;

        } elseif (
          Jogo.controleTecla[Jogo.Tecla.DIREITA.ordinal()]) {
            dx = 1;
        }

        if (dx != 0) {
            dy = 0;
            moveu = true;
        }

    } else if (dx != 0) {
        if (Jogo.controleTecla[Jogo.Tecla.CIMA.ordinal()]) {
            dy = -1;
        } else if (
          Jogo.controleTecla[Jogo.Tecla.BAIXO.ordinal()]) {
            dy = 1;
        }

        if (dy != 0) {
            dx = 0;
            moveu = true;
        }
    }
```

```
}
```

Utilizamos `dx` e `dy` para controlar a direção da serpente, e essas variáveis podem ter o valor `0` (sem direção), `-1` (para a esquerda ou para cima) e `1` (para a direita ou para baixo). Mas sempre que `dx` for diferente de zero, `dy` será igual a zero, e vice-versa.

Se o jogador não tiver feito nenhum movimento (`moveu` igual a falso), verificamos se a serpente está se movendo na vertical (`dy` diferente de zero). Então verificamos se o movimento do jogador foi para a esquerda ou direita, atualizando `dx`. Tendo um movimento válido, zeramos `dy` e marcamos que o movimento foi realizado. O mesmo ocorre para a verificação na horizontal, com suas respectivas direções, atualizando `dy`.

Durante o jogo, verificamos se a cabeça da serpente saiu da tela, se ela colidiu com algum elemento do cenário ou se colidiu com o próprio rastro. Se isso ocorrer, inativamos a serpente para ignorar as outras validações, e alteramos o estado do jogo para informar ao jogador que ele perdeu.

```
if (Util.saiu(serpente, largura, altura)) {
    serpente.setAtivo(false);
    estado = Estado.PERDEU;

} else {
    // colisão com cenário
    for (int i = 0; i < contadorNivel; i++) {
        if (Util.colide(serpente, nivel[i])) {
            serpente.setAtivo(false);
            estado = Estado.PERDEU;
            break;
        }
    }

    // colisão com o rastro
    for (int i = 0; i < contadorRastro; i++) {
        if (Util.colide(serpente, rastros[i])) {
            serpente.setAtivo(false);
            estado = Estado.PERDEU;
            break;
        }
```

```
    }
}
```

Mas nem tudo é derrota. Se a serpente colidir com o elemento `fruta`, voltamos o `temporizador` (para simular uma pequena pausa), aumentamos a contagem de rastros (`contadorRastro`), e inativamos o elemento fruta para que reapareça em outro lugar. Já se a contagem de rastros chegar ao número máximo, o jogador ganha o jogo.

```
if (Util.colide(fruta, serpente)) {
    // Adiciona uma pausa
    temporizador = -10;
    contadorRastro++;
    fruta.setAtivo(false);

    if (contadorRastro == rastros.length) {
        serpente.setAtivo(false);
        estado = Estado.GANHOU;
    }
}
```

Adicionada de forma randômica, a fruta é posicionada na tela, dividindo a largura e altura da `tela` pela largura e altura do elemento. Para uma tela do tamanho de 450px, com largura e altura padrão de 25px (`_LARG`), temos um total de 18 posições possíveis. Dessa forma estamos trabalhando com a tela como se ela fosse um array.

```
// Adicionando frutas
if (estado == Estado.JOGANDO && !fruta.isAtivo()) {
    int x = rand.nextInt(largura / _LARG);
    int y = rand.nextInt(altura / _LARG);

    fruta.setPx(x * _LARG);
    fruta.setPy(y * _LARG);
    fruta.setAtivo(true);

    // colisão com a serpente
    if (Util.colide(fruta, serpente)) {
        fruta.setAtivo(false);
```

```
            return;
        }

        // colisão com rastro
        for (int i = 0; i < contadorRastro; i++) {
            if (Util.colide(fruta, rastros[i])) {
                fruta.setAtivo(false);
                return;
            }
        }

        // colisão com cenário
        for (int i = 0; i < contadorNivel; i++) {
            if (Util.colide(fruta, nivel[i])) {
                fruta.setAtivo(false);
                return;
            }
        }

}
```

Se, por acaso, a fruta na hora em que for adicionada colidir com algum elemento, ela será desativada e tentamos no próximo loop, quando o método `atualizar()` será chamado novamente. Depois de tudo que foi carregado e atualizado, só nos falta desenhar:

```
@Override
public void desenhar(Graphics2D g) {

    if (fruta.isAtivo()) {
        fruta.desenha(g);
    }

    for(Elemento e : nivel){
        if(e == null)
            break;

        e.desenha(g);
    }
```

```
    for (int i = 0; i < contadorRastro; i++) {
        rastros[i].desenha(g);
    }

    serpente.desenha(g);
```

A novidade aqui é que percorremos o array `nivel` sem utilizar o `contadorNivel`, parando quando encontramos um elemento nulo.

```
texto.desenha(g,
                String.valueOf(rastros.length - contadorRastro),
                largura - 35, altura);

    if (estado != Estado.JOGANDO) {

        if (estado == Estado.GANHOU)
            texto.desenha(g, "Ganhou!", 180, 180);
        else
            texto.desenha(g, "Vixe!", 180, 180);
    }

    if (Jogo.pausado)
        Jogo.textoPausa.desenha(g, "PAUSA", largura / 2 -
                    Jogo.textoPausa.getFonte().getSize(),
                    altura / 2);
}
```

Utilizamos o objeto `texto` (da nossa classe `Texto` que estende de `Elemento`) para mostrar na tela quantas frutas faltam para o jogador ganhar o jogo. Se ele ganhar ou perder, usamos o método `desenha` para exibir a mensagem de vitória ou derrota e, enquanto o jogo estiver pausado, exibimos a mensagem de pausa na tela.

4.3 Codificando o jogo

Esperando que você traga novos níveis, nosso código encontra-se em https://github.com/logicadojogo/fontes/tree/master/Cap04.

Gostaria de ter colocado o código completo da classe `Nivel.java` nas páginas deste livro, mas eles perderiam a formatação e a legibilidade.

Nossa caixa de ferramentas (código base) não sofreu alteração, e modificamos nossa classe `Jogo.java` para que a janela se ajuste à largura da tela.

Para finalizar, com os conhecimentos adquiridos aqui, já conseguimos criar as barreiras que faltaram na nossa versão do Space Invader no capítulo 2.

Não fizemos aqui

- Ir para o próximo nível quando o jogador ganhar (até completar todos e ganhar o jogo);
- Definir a posição inicial da serpente de acordo com a fase.

Melhore você mesmo

- Faça cenários com mais cores;
- Crie seu próprio nível.

4.4 Resumo

Vimos como as fases de um jogo podem ser criadas e como o cenário pode ser construído para trabalhar com elas. Mas não precisamos nos limitar apenas à construção de níveis, elementos do jogo podem ser criados dessa forma, como veremos no próximo capítulo.

Mesmo utilizando poucas linhas e colunas, a diversidade de levels que conseguimos criar é grande. Este é um jogo onde os tamanhos (tela, nível e elemento) fazem diferença. Estamos utilizando valores fixos (tela 450x450, nível 18x18 e elemento 25x25), mas estes estão ligados, então, ao alterar algum deles, teríamos de modificar os outros, seja dividindo ou multiplicando um pelo o outro.

Podemos selecionar na introdução a fase e a velocidade do jogo, nada que não tenhamos visto no capítulo anterior, e construímos o jogo introduzindo o fluxo de vitória e derrota.

Capítulo 5

Tetris ou Quatro Quadrados

Posso dizer que Tetris é o jogo russo mais famoso já criado. Mesmo não conseguindo evitar a guerra fria, e quase causando uma guerra entre a Atari e a Nintendo, o jogo de Alekey Pajitnov, criado em 1984, foi pulicado pela Atari (versão não oficial desenvolvida pela Tengen) e pela Nintendo para NES, em 1989 com um mês de diferença.

Além de criarmos este quebra-cabeças animado e viciante capaz de garantir algumas boas horas de diversão, veremos neste capítulo:

- Manipulações avançadas de arrays, como colisão e rotação;
- Música e efeitos sonoros;
- Calcular o tamanho do desenho de acordo com a tela.

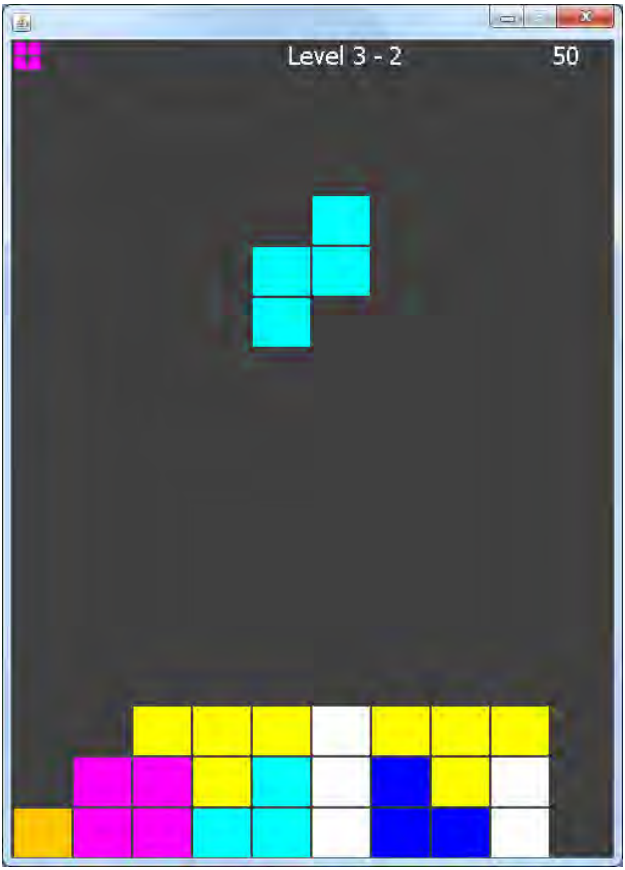

Fig. 5.1: O jogo

5.1 Um jogo de arrays

Vimos no capítulo anterior como usar um array tridimensional para criar fases, mas dessa vez vamos além: das peças que se movem na tela até a grade onde elas são encaixadas. Usaremos arrays, deixando nossos objetos da classe `Elemento` fora do jogo, ao menos neste.

Temos a seguir todo o esplendor que um array com três dimensões deve ter:

```
public class Peca {
```

```
    public static Color[] Cores = {
            Color.GREEN, Color.ORANGE,
            Color.YELLOW, Color.CYAN, Color.BLUE,
            Color.MAGENTA, Color.WHITE };

    public static final int[][][] PECAS = {
            {
                { 0, 1, 0 },
                { 0, 1, 0 },
                { 1, 1, 0 } },
            {
                { 0, 1, 0 },
                { 0, 1, 0 },
                { 0, 1, 1 } },
            {
                { 1, 1, 1 },
                { 0, 1, 0 },
                { 0, 0, 0 } },
            {
                { 1, 0, 0 },
                { 1, 1, 0 },
                { 0, 1, 0 } },
            {
                { 0, 0, 1 },
                { 0, 1, 1 },
                { 0, 1, 0 } },
            {
                { 1, 1 },
                { 1, 1 } },
            {
                { 0, 1, 0, 0 },
                { 0, 1, 0, 0 },
                { 0, 1, 0, 0 },
                { 0, 1, 0, 0 } }
    };
}
```

Essa é nossa classe `Peca.java`, responsável por conter nossas peças e as cores delas, dentro de (mais) um array. Não existe nenhum motivo especial

para o tipo do nosso array `PECAS` ser um inteiro, mas note que cada peça tem o mesmo número de linhas e colunas, sendo que os zeros representam os espaços em branco das peças.

Cada peça poderia ser construída com um número que representasse sua cor, por exemplo, `1` para cor verde, `2` para cor laranja, e assim por diante. Entretanto, nosso código ficou mais simples assim, e também é mais fácil mudar a cor da peça.

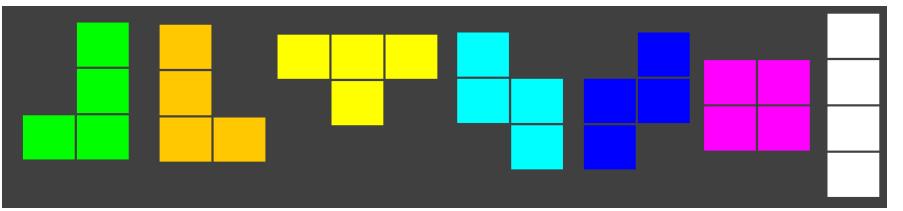

Fig. 5.2: Peças na ordem em que aparecem no código

Nossa classe `JogoCenario.java` contém uma grade que representa os espaços onde as peças podem ser encaixadas. Ela é do mesmo tipo do nosso array de peças, contendo 10 colunas e 16 linhas, sendo que cada espaço na grade pode conter um desses valores: `-1` para os espaços vazios; `-2` para as linhas completas; ou o índice que representa alguma das sete peças (`0` até `6`).

```
...
private static final int ESPACAMENTO = 2;

private static final int ESPACO_VAZIO = -1;

private static final int LINHA_COMPLETA = -2;

private int largBloco, altBloco; // largura bloco e altura bloco

private int ppx, ppy; // Posição peça x e y

private final int[][] grade = new int[10][16];

...
```

```
private boolean animar;
private boolean depurar;
...
```

Essas são algumas das diversas variáveis que utilizamos no jogo e veremos com mais detalhes ao longo do capítulo.

Algumas linhas completas depois, em nosso método `carregar`:

```
@Override
public void carregar() {
    largBloco = largura / grade.length;
    altBloco = altura / grade[0].length;

    for (int i = 0; i < grade.length; i++) {
        for (int j = 0; j < grade[0].length; j++) {
            grade[i][j] = ESPACO_VAZIO;
        }
    }

...
       adicionaPeca();
}
```

Consideramos que as peças são formadas por blocos, quatro cada uma, e cada bloco tem sua largura e altura baseados no tamanho disponível de cada espaço na grade. Esses valores são armazenados em `largBloco` e `altBloco`, que nada mais é que a divisão da `largura` da `tela` pelo tamanho do array (500 / 10), e a `altura` pelo tamanho do array interno (672 / 16).

Iniciamos nossa `grade` com o valor `ESPACO_VAZIO`, logo depois uma peça, escolhida aleatoriamente, é adicionada ao jogo.

```
public void adicionaPeca() {

    ppy = -2;
    ppx = grade.length / 2 - 1;

    // Primeira chamada
```

```
    if (idPeca == -1)
        idPeca = rand.nextInt(Peca.PECAS.length);
    else
        idPeca = idPrxPeca;

    idPrxPeca = rand.nextInt(Peca.PECAS.length);

    // Isso acontece muito
    if (idPeca == idPrxPeca)
        idPrxPeca = rand.nextInt(Peca.PECAS.length);

    peca = Peca.PECAS[idPeca];
    corPeca = Peca.Cores[idPeca];

}
```

A peça começa um pouco acima do topo da tela e próxima do centro. Usamos `idPeca` e `peca` para controlar a peça atual (que o jogador controla), e `idPrxPeca` para a próxima peça, que precisamos saber de antemão para dizer ao jogador que ela virá a seguir. Na primeira chamada, tanto a peça atual quanto a que virá em seguida são escolhidas aleatoriamente.

Da segunda chamada em diante, o valor da próxima peça passa para a atual e a próxima recebe um novo valor. Caso ocorra de ambas serem iguais, tentamos outra vez. Como estamos trabalhando com apenas 7 possibilidades, isso ocorre bastante.

Durante o jogo, percorremos o array da peça selecionada, comparando seus blocos diferentes de zero com o valor correspondente na grade. Utilizamos `ppx` e `ppy` (posição da peça no eixo X e Y) para movimentá-la. Quando ela colide com a base da grade ou com outra peça, a grade fica com o mesmo valor do `idPeca` (0 a 6), nos locais equivalentes aos valores diferentes de 0 da peça.

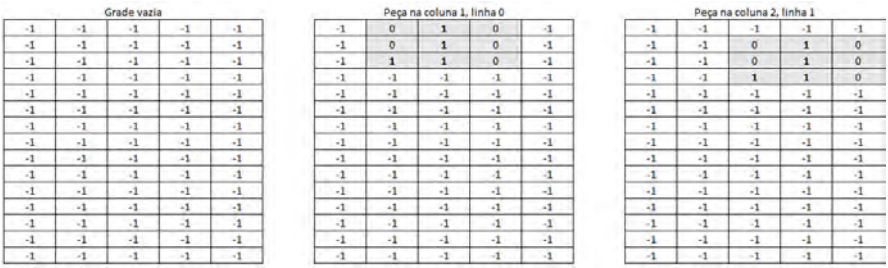

Fig. 5.3: Simulação Grade e Peça

Considerando a figura 5.3, temos primeiramente a grade vazia, depois a grade com uma peça posicionada na coluna 1 e linha 0, onde `ppx` é 1 e `ppy` 0. Depois, a peça é movida uma coluna para a direita e uma linha para baixo, onde agora `ppx` é 2 e `ppy` 1.

Isso quer dizer que, além de percorrermos o array, somamos seu índice com `ppx` e `ppy` para sabermos a posição da peça. O método que usamos para adicionar a peça na grade é um bom exemplo de como é o relacionamento entre grade e peça:

```
private void adicionarPecaNaGrade() {
    for (int col = 0; col < peca.length; col++) {
        for (int lin = 0; lin < peca[col].length; lin++) {
            if (peca[lin][col] != 0) {
                grade[col + ppx][lin + ppy] = idPeca;

            }
        }
    }
}
```

Percorremos o array da peça e, para cada valor diferente de 0, a grade tem seu valor alterado de -1 para o valor do `idPeca`. Então, saberemos com qual cor aquele bloco deverá ser pintado. Se os valores na grade formarem uma linha completa (todos os valores na linha forem diferentes de -1), ela passa a ter o valor de `LINHA_COMPLETA`.

Dica

Como o que determina o que é linha e o que é coluna é a forma como percorremos e desenhamos o array na tela, é muito fácil nos confundirmos com as variáveis na iteração do loop. Para minimizar essa confusão, adotamos a seguinte regra:

No laço, chamamos abreviadamente de `col` as variáveis que interagem no eixo X, e de `lin` as que interagem com o eixo Y.

5.2 Desenhando arrays

Agora, como a peça na grade é desenhada? Já vimos que o tamanho da tela dividido pelo tamanho da grade é quem determina a largura e altura da peça (`largBloco` e `altBloco`).

```
for (int col = 0; col < grade.length; col++) {
    for (int lin = 0; lin < grade[0].length; lin++) {
        int valor = grade[col][lin];

        if (valor == ESPACO_VAZIO)
            continue;

        if (valor == LINHA_COMPLETA)
            g.setColor(Color.RED);
        else
            g.setColor(Peca.Cores[valor]);

        int x = col * largBloco + ESPACAMENTO;
        int y = lin * altBloco + ESPACAMENTO;

        g.fillRect(x, y, largBloco - ESPACAMENTO,
                   altBloco - ESPACAMENTO);

    }
}
```

Percorremos a grade obtendo e verificando o valor de cada linha e coluna. Se o `valor` for um espaço vazio, ignoramos; caso contrário, definimos com qual cor a peça será pintada. Utilizamos a cor vermelha para pintarmos as linhas que estiverem completas.

Depois, achamos a posição do bloco na tela e, de forma similar ao capítulo anterior, pintamos nosso bloco levando em conta o valor do `ESPACAMENTO`. O espaçamento é opcional, serve apenas para deixar um vão entre as peças.

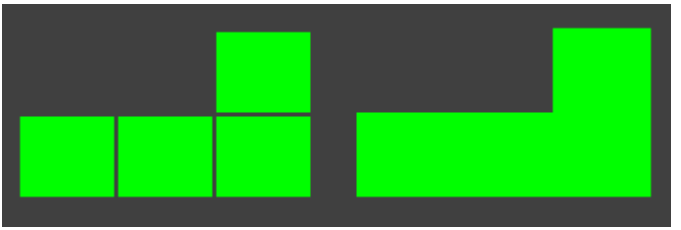

Fig. 5.4: Peça com e sem espaçamento

Dica

Aumentar ou diminuir exageradamente o espaçamento entre as peças deixa o jogo com efeitos no mínimo interessantes.

O código para desenhar a peça em jogo é bem parecido com o anterior, levando em consideração a posição da peça nos eixos:

```
if (peca != null) {
    g.setColor(corPeca);

    for (int col = 0; col < peca.length; col++) {
        for (int lin = 0; lin < peca[col].length; lin++) {
            if (peca[lin][col] != 0) {

                int x = (col + ppx) * largBloco + ESPACAMENTO;
                int y = (lin + ppy) * altBloco + ESPACAMENTO;
```

```
                g.fillRect(x, y, largBloco - ESPACAMENTO,
                            altBloco - ESPACAMENTO);

            } else if (depurar) {
                g.setColor(Color.PINK);
                int x = (col + ppx) * largBloco + ESPACAMENTO;
                int y = (lin + ppy) * altBloco + ESPACAMENTO;

                g.fillRect(x, y, largBloco - ESPACAMENTO,
                            altBloco - ESPACAMENTO);

                g.setColor(corPeca);
            }
        }
    }
}
```

Para ajudar na depuração do código, se a variável `depurar` for verdadeira, exibimos as partes ocultas da peça (os valores iguais a `0`) na cor rosa. Para ajudar na tomada de decisão do jogador, desenhamos a próxima peça usando um tamanho menor que a largura do bloco.

```
int miniatura = largBloco / 4;
int[][] prxPeca = Peca.PECAS[idPrxPeca];
g.setColor(Peca.Cores[idPrxPeca]);

for (int col = 0; col < prxPeca.length; col++) {
    for (int lin = 0; lin < prxPeca[col].length; lin++) {
        if (prxPeca[lin][col] == 0)
            continue;

        int x = col * miniatura + ESPACAMENTO;
        int y = lin * miniatura + ESPACAMENTO;

        g.fillRect(x, y, miniatura - ESPACAMENTO,
                    miniatura - ESPACAMENTO);

    }
}
```

Multiplicamos coluna e linha pelo tamanho da `miniatura`, sem esquecer de adicionar o espaçamento para o formato do desenho ficar parecido.

> **Nota**
>
> Na sala de informática do colégio, após descobrir que um professor havia criado sua própria versão do Tetris (em Pascal, se não me engano), esse jogo passou a me intrigar mais do que divertir.
>
> Eu simplesmente não conseguia entender como as peças se encaixavam umas nas outras, como elas ficavam presas, e depois simplesmente formavam linhas que desapareciam. Bem, espero que, ao final deste capítulo, você entenda como isso funciona melhor que eu, ou pelo menos, tão bem quanto o hacker Vadim Gerasimov, responsável pelo porte do jogo para IBM-PC.

5.3 Jogando o código

Quando o jogador move a peça para a esquerda, para a direita ou para baixo, precisamos validar o movimento e garantir que a nova posição não quebre o jogo. Nosso método `validaMovimento` garante que isso não aconteça.

```
public boolean validaMovimento(int[][] peca, int px, int py) {

    if (peca == null)
        return false;

    for (int col = 0; col < peca.length; col++) {
        for (int lin = 0; lin < peca[col].length; lin++) {
            if (peca[lin][col] == 0)
                continue;

            int prxPx = col + px; // Próxima posição peça x
            int prxPy = lin + py; // Próxima posição peça y

            if (prxPx < 0 || prxPx >= grade.length)
                return false;
```

```
            if (prxPy >= grade[0].length)
                return false;

            if (prxPy < 0)
                continue;

            // Colidiu com uma peça na grade
            if (grade[prxPx][prxPy] > ESPACO_VAZIO)
                return false;

        }
    }

    return true;
}
```

Como de costume, ignoramos os valores iguais a `0`. Depois de acharmos a próxima posição da peça, verificamos se ela não acaba fora da tela, sendo menor que zero ou maior que o tamanho da grade. Isso para validação horizontal; já na vertical a peça pode começar fora da tela. Então, somente verificamos se a futura posição não ultrapassa o tamanho da grade, ignorando um valor menor que zero.

Por fim, verificamos se a posição coincide com um valor já preenchido na grade. Caso isso ocorra, será um movimento inválido, já que uma peça não pode se sobrepor a outra. Observe que a ordem da validação é importante e usamos a mesma validação para girar a peça.

Verificar colisão é um método semelhante ao de validar movimento, tão semelhante que poderia ser o mesmo método com algumas alterações. Entretanto, separamos em dois para deixar claro quando estamos validando o movimento e quando estamos verificando se a peça colidiu.

```
private boolean colidiu(int px, int py) {

    if (peca == null)
        return false;

    for (int col = 0; col < peca.length; col++) {
```

```
        for (int lin = 0; lin < peca[col].length; lin++) {
            if (peca[lin][col] == 0)
                continue;

            int prxPx = col + px;
            int prxPy = lin + py;

            // Chegou na base da grade
            if (prxPy == grade[0].length)
                return true;

            // Fora da grade
            if (prxPy < 0)
                continue;

            // Colidiu com uma peça na grade
            if (grade[prxPx][prxPy] > ESPACO_VAZIO)
                return true;
        }
    }

    return false;
}
```

Neste método, não recebemos a `peca` como parâmetro, uma vez que sempre verificamos a colisão da peça em jogo. Mas a maior diferença é que retornamos verdadeiro onde antes era retornado falso, além de não validarmos a posição da peça na horizontal. Esse método poderia facilmente ser substituído por:

```
private boolean colidiu(int px, int py) {
    return !validaMovimento(peca, px, py);
}
```

Mais simples que validar o movimento ou saber se a peça colidiu é verificar se ela ficou fora da grade:

```
private boolean parouForaDaGrade() {
    if (peca == null)
```

```
            return false;

        for (int lin = 0; lin < peca.length; lin++) {
            for (int col = 0; col < peca[lin].length; col++) {
                if (peca[lin][col] == 0)
                    continue;
                // Fora da grade
                if (lin + ppy < 0)
                    return true;
            }
        }

        return false;
    }
```

Se a peça `colidiu`, nós a adicionamos na grade e verificamos se ela preencheu uma ou mais linhas. Se isso ocorrer, marcamos as linhas preenchidas:

```
private boolean marcarLinha() {
    int multPontos = 0;

    for (int lin = grade[0].length - 1; lin >= 0; lin--) {
        boolean linhaCompleta = true;

        for (int col = grade.length - 1; col >= 0; col--) {
            if (grade[col][lin] == ESPACO_VAZIO) {
                linhaCompleta = false;
                break;
            }
        }

        if (linhaCompleta) {
            multPontos++;
            for (int col = grade.length - 1; col >= 0; col--) {
                grade[col][lin] = LINHA_COMPLETA;
            }
        }
    }
```

Este `loop` já difere dos anteriores. Repare bem que, além de começarmos pelas linhas, começamos da última para a primeira.

Começamos confiantes, acreditando que acharemos uma linha completa. Então, linha por linha, verificamos cada coluna, mas ao encontrar uma coluna vazia, falamos que o sonho da linha completa não está naquela linha (`linhaCompleta = false`), e ignoramos as demais colunas, passando para a linha seguinte.

Entretanto, quando todas as colunas estão preenchidas, incrementamos nosso multiplicador de pontos e percorremos a mesma linha novamente, marcando cada coluna (`grade[col][lin] = LINHA_COMPLETA`).

```
pontos += multPontos * multPontos;
    linhasFeistas += multPontos;

    if (nivel == 9 && linhasFeistas >= 9) {
        estado = Estado.GANHOU;

    } else if (linhasFeistas >= 9) {
        nivel++;
        linhasFeistas = 0;
    }

    return multPontos > 0;
}
```

Depois de verificarmos linhas e colunas da grade, premiamos o jogador com seus pontos, e quanto mais linhas consecutivas, mais pontos. Os níveis do jogo vão de 1 a 9, fazendo as peças caírem mais rápido conforme vão aumentando. Toda vez que o jogador completa 9 ou mais linhas, o nível do jogo sobe automaticamente e, quando ele completa a última linha do último nível, ganha o jogo.

Precisamos saber se marcamos alguma linha, então, retornamos `multPontos > 0`, que será verdadeiro se tivermos alguma linha completa. Tendo linhas completas, fazemo-las desaparecem, e as que estavam acima delas descerem.

```
for (int col = 0; col < grade.length; col++) {
    for (int lin = grade[0].length - 1; lin >= 0; lin--) {
```

```
        if (grade[col][lin] == LINHA_COMPLETA) {
            int moverPara = lin;
            int prxLinha = lin - 1;

            for (; prxLinha > -1; prxLinha--) {
                if (grade[col][prxLinha] == LINHA_COMPLETA)
                    continue;
                else
                    break;
            }

            for (; moverPara > -1; moverPara--, prxLinha--) {

                if (prxLinha > -1)
                    grade[col][moverPara] = grade[col][prxLinha];
                else
                    grade[col][moverPara] = ESPACO_VAZIO;

            }
        }
    }
}
```

Este é o código principal do método `descerColunas()`, e nosso foco agora é mais nas colunas do que nas linhas. Verificamos verticalmente cada coluna, da última para a primeira, e os valores contidos em cada, já que uma coluna pode ter mais blocos que outra. Quando achamos uma posição marcada como `LINHA_COMPLETA`, definimos que esta será a nova base das peças acima (`int moverPara = lin`).

Como o jogador pode fazer até quatro linhas consecutivas, usamos `prxLinha` para contá-las. Sendo que podemos ter linhas completas seguidas ou intermitentes, a verificação se estende até a primeira coluna (laço interno enquanto `moverPara` for maior que `-1`). Enquanto `prxLinha` tiver um valor válido, sabemos que temos blocos para descer; se não, os blocos alocados naquele espaço já desceram, deixando um espaço vazio.

Nota

Eu acho este um dos métodos mais complexos do jogo, a ponto de quase chamar o capítulo de *Tetris, o jogo do* `ArrayIndexOutOfBoundsException`.

Finalmente giramos a peça, e temos duas versões para este método. A primeira não realoca a peça, então se ela estiver perto demais das laterais, o jogador não conseguirá girá-las, igual a versões do Tetris para NES (diferente da versão da Tengen, que tinha até modo multijogador). Simplesmente usamos um array temporário e colocamos em linha os valores que estão em coluna. Por isso, os arrays das peças precisam ter o mesmo número de linhas e colunas.

```
protected void girarPeca(boolean sentidoHorario) {
    if (peca == null)
        return;

    final int[][] temp = new int[peca.length][peca.length];

    for (int i = 0; i < peca.length; i++) {
        for (int j = 0; j < peca.length; j++) {
            if (sentidoHorario)
                temp[j][peca.length - i - 1] = peca[i][j];
            else
                temp[peca.length - j - 1][i] = peca[i][j];
        }
    }

    System.out.println("Antes:");
    imprimirArray(peca);
    System.out.println("Depois:");
    imprimirArray(temp);

    if (validaMovimento(temp, ppx, ppy)) {
        peca = temp;
    }
```

```
}
```

Se o jogador girar a peça no sentido horário, os valores da primeira linha vão para a última coluna, e da última linha para a primeira coluna. Se girar no sentido anti-horário, os valores da primeira linha vão para a primeira coluna.

A solução é simples, mas a execução pode parecer confusa caso não consiga visualizar as posições do array. Logo, fazer um passo a passo no papel vai ajudar. Veja nossa simulação com uma peça fictícia:

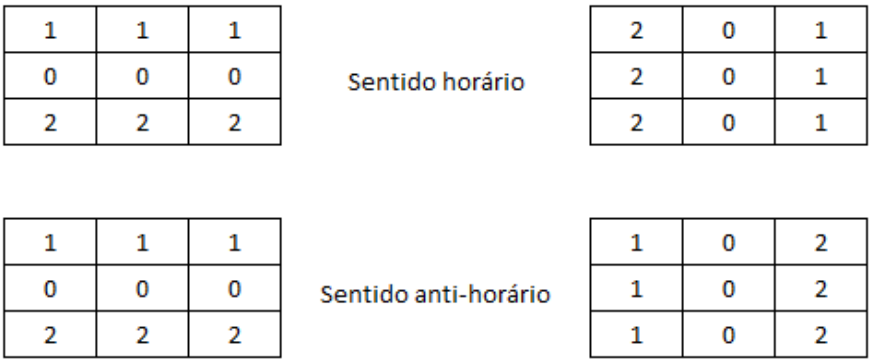

Fig. 5.5: Ciranda das peças

Você ainda pode contar com o `depurar` e com o nosso método `imprimirArray`, capazes de produzir esse resultado:

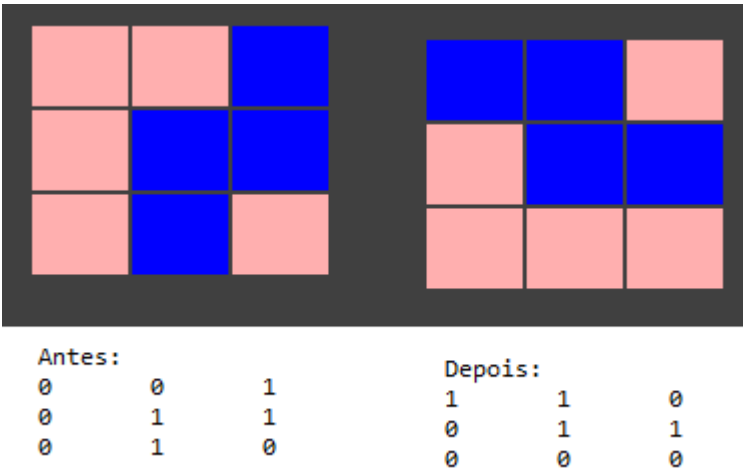

Fig. 5.6: Girando a peça

Antes de seguir em frente com o movimento, verificamos se a peça virada está em uma posição válida, mas se a peça girada acabar fora da tela, as modificações serão ignoradas.

A segunda versão do método resolve esse problema, reposicionando a peça:

```
private void girarReposicionarPeca(boolean sentidoHorario) {
    if (peca == null)
        return;

    int tempPx = ppx;
    final int[][] tempPeca = new int[peca.length][peca.length];

    //... Girar a peça

    // Reposiciona peça na tela
    for (int i = 0; i < tempPeca.length; i++) {
        for (int j = 0; j < tempPeca.length; j++) {
            if (tempPeca[j][i] == 0) {
                continue;
            }
```

```
            int prxPx = i + tempPx;

            if (prxPx < 0)
                tempPx = tempPx - prxPx;

            else if (prxPx == grade.length)
                tempPx = tempPx - 1;
        }
    }

    if (validaMovimento(tempPeca, tempPx, ppy)) {
        peca = tempPeca;
        ppx = tempPx;
    }
}
```

Se a peça acabar fora da grade, teremos um `prxPx` negativo ou maior que o número de colunas. O valor sendo negativo, subtraímos o mesmo valor gerando um resultado positivo. Caso o valor seja maior que o esperado, subtraímos 1 de `tempPx`, até que todos os blocos estejam dentro dos limites da tela.

Utilizamos uma variável temporária, já que as alterações só podem acontecer se a peça estiver em uma posição válida, neste caso, não colidindo com outras peças.

Na hora de atualizar o jogo, nada muito diferente do que já fizemos nos outros:

```
...
if (Jogo.controleTecla[Jogo.Tecla.ESQUERDA.ordinal()]) {
    if (validaMovimento(peca, ppx - 1, ppy))
        ppx--;

} else if (Jogo.controleTecla[Jogo.Tecla.DIREITA.ordinal()]) {
    if (validaMovimento(peca, ppx + 1, ppy))
        ppx++;
}
```

```
if (Jogo.controleTecla[Jogo.Tecla.CIMA.ordinal()]) {
    girarReposicionarPeca(false);

} else if (Jogo.controleTecla[Jogo.Tecla.BAIXO.ordinal()]) {
    if (validaMovimento(peca, ppx, ppy + 1))
        ppy++;
}
...
```

Para esquerda ou para direita, decrementamos ou incrementamos `ppx` em 1, movendo a peça coluna por coluna. Ao pressionar para baixo, aceleramos a descida da peça aumentando `ppy`, uma linha por vez, e ao pressionar para cima, giramos a peça. Lembrando que a atualização desses valores só ocorre se a peça tiver um movimento válido.

Outro destaque do método `atualizar()` é que, depois de o jogador formar alguma linha completa, não adicionamos uma nova peça imediatamente, dando um tempo para ele ver suas linhas completas destacadas na cor vermelha antes de desaparecerem, uma pequena animação.

```
if (animar && temporizador >= 5) {
    animar = false;

    descerColunas();
    adicionaPeca();

} else if (temporizador >= 20) {
    temporizador = 0;

    if (colidiu(ppx, ppy + 1)) {

...
        if (!parouForaDaGrade()) {
            adicionarPecaNaGrade();
            animar = marcarLinha();

            peca = null;

            if (!animar)
```

```
                adicionaPeca();

            } else {
                estado = Estado.PERDEU;
            }

        } else
            ppy++;

} else
    temporizador += nivel;
```

Usamos o velho e conhecido `temporizador` para saber se chegou a hora de animar ou atualizar o jogo. Na hora de atualizar, verificamos se a próxima posição da peça resultará em uma colisão. Caso não colida, continuamos descendo a peça, de acordo com a velocidade do nível do jogo. Caso colida, verificamos se algum bloco parou fora da grade; se isso acontecer, o jogo acaba, se não, adicionamos a peça na grade e verificamos se alguma linha completa foi formada. Então, `animar` recebe o valor verdadeiro e as linhas são marcadas e exibidas em outra cor, daí apagamos a peça (`peca = null`) para que ela não seja mais desenhada.

Se nenhuma linha completa foi formada, `animar` recebe o valor falso, então adicionamos uma nova peça. O nosso `animar` não tem animação, apenas esperamos um pouco antes de sumir com as linhas marcadas e adicionarmos uma nova peça, mas encorajo você a fazer algo mais interessante aqui.

Dica

Para ajudar nos testes do jogo, adicionamos uma tecla para trocar a peça principal. Ficou assim:

```
if (depurar && Jogo.controleTecla[Jogo.Tecla.BC.ordinal()]){
    if (++idPeca == Peca.PECAS.length)
        idPeca = 0;

    peca = Peca.PECAS[idPeca];
    corPeca = Peca.Cores[idPeca];
}
```

Temos aqui uma boa versão do Tetris, sendo um dos jogos mais completos que fizemos até agora: com início, meio e fim; desafios que aumentam gradualmente; cores, pausa e pontuação. Mas sinceramente, só isso não é o suficiente, e um jogo sempre fica melhor com som.

5.4 Efeitos sonoros

O Java tem suporte nativo aos formatos `wav`, `aif`, `rmf`, `au` e `mid`. Estes podem variar dependendo da plataforma, e outros são suportados usando bibliotecas à parte, como o Java Media Framework (JMF), onde você consegue reproduzir desde o popular `mp3` até formatos de vídeo.

Aqui veremos os formatos `.mid` (*Musical instrument Digital Interface*) e `.wav` (*Windows Wave*).

Começamos pelo objeto `Clip` do pacote `javax.sound.sampled`.

```
import javax.sound.sampled.AudioInputStream;
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.Clip;

try {
    AudioInputStream as = AudioSystem.getAudioInputStream(
                    new File("som/piano_quebrado.mid"));
```

```
    Clip clip = AudioSystem.getClip();
    clip.open(as);

} catch (Exception e) {
    e.printStackTrace();
}
```

Primeiro, carregamos o arquivo `piano_quebrado.mid`, que está dentro da pasta `som`, na pasta raiz do projeto. Depois, obtemos um objeto `Clip` para controlar o áudio carregado. Tudo isso tratando qualquer exceção que possa ocorrer, já que ainda poderemos jogar sem som.

Basicamente, o que fazemos com um objeto `Clip` é tocar uma vez, várias vezes ou continuamente o áudio e, claro, parar de tocar. Ao chamar `clip.start()` ou `clip.loop(0)`, tocamos o áudio uma vez. Já `clip.loop(10)` toca o áudio 11 vezes (toca uma vez e repete dez).

Para tocar até não querer mais, utilize `clip.loop(Clip.LOOP_CONTINUOUSLY)` e, para não tocar mais, `clip.stop()`. Quando descarregar a cena, não se esqueça de parar a música e liberar recursos.

```
@Override
publicvoid descarregar() {

    if (clip != null) {
        clip.stop();
        clip.close();
    }
}
```

Com Java, você tem bastante controle sobre o áudio que está sendo reproduzido. Recomendo procurar por Java Sound API: Java Sound Demo (http://www.oracle.com/technetwork/java/index-139508.html) , onde você consegue ver exemplos como:

- Tocar formatos `wav`, `aif`, `rmf`, `au` e `mid`;
- Controlar o volume de áudio, inclusive individualmente em cada canal (*Panning*);

- Gravação de áudio;
- Gerar seu próprio `mid` com um simulador de piano (com vários efeitos, inclusive de som de tiro);
- Fazer solos de bateria.

> **Dica**
>
> Foi do Java Sound Demo que veio meu MIDI de sucesso: `piano_quebrado`.

Quando falamos de MIDI, o objeto `Clip` é para o mais básico, uma vez que, utilizando `Sequencer`, podemos manipular arquivos MIDI com recursos adicionais, como acelerar o áudio alterando as batidas por minuto. Veja mais em http://docs.oracle.com/javase/tutorial/sound/MIDI-seq-intro.html.

Para carregar nosso `.mid` utilizando `Sequencer`, nosso código ficaria assim:

```
import javax.sound.midi.MidiSystem;
import javax.sound.midi.Sequencer;

try {
    Sequencer seq = MidiSystem.getSequencer();
    seq.setSequence(
        MidiSystem.getSequence(
            new File("som/piano_quebrado.mid")));
    seq.open();
} catch (Exception e) {
    e.printStackTrace();
}

seq.setLoopCount(Sequencer.LOOP_CONTINUOUSLY);
seq.start();
```

As diferenças no código começam no pacote. Usamos `javax.sound.midi` em vez de `javax.sound.sampled`, e a classe

`MidiSystem` em vez da classe `AudioSystem`. Note que utilizamos o atalho `setSequence`, mesmo assim o método `open` deve ser chamado. Não podemos nos esquecer de chamar o método `start`, já que não temos o atalho `loop`.

O restante do código fica parecido, apenas atenção para não confundir o objeto `Sequence` com a interface `Sequencer` (que termina com R).

Supondo que você queira acelerar o áudio quando o jogador estiver com linhas próximas ao topo, ou deixar a velocidade do áudio de acordo com o nível das fases, bastaria chamar `seq.setTempoInBPM(300)`, que, neste caso, acelera a música em `300` batidas por minuto.

5.5 Tetris sonoro

Além de uma música MIDI de fundo, adicionamos dois efeitos (`adiciona_peca.wav` e `109662_grunz_success.wav`): um para quando a peça colidir, e outro quando o jogador completar uma ou mais linhas.

```
// Som
private AudioInputStream as;
private Clip clipAdicionarPeca;
private Clip clipMarcarLinha;
private Sequencer seqSomDeFundo;

...

@Override
public void carregar() {

    try {
        as = AudioSystem.getAudioInputStream(
                new File("som/adiciona_peca.wav"));
        clipAdicionarPeca = AudioSystem.getClip();
        clipAdicionarPeca.open(as);

        as = AudioSystem.getAudioInputStream(
                new File("som/109662_grunz_success.wav"));
```

```
        clipMarcarLinha = AudioSystem.getClip();
        clipMarcarLinha.open(as);
```

Utilizamos dois objetos `Clip`, `clipAdicionarPeca` e `clipMarcarLinha`, para nosso áudio WAV. Um efeito sonoro para quando adicionamos a peça na grade e outro para quando marcamos as linhas feitas. Ficamos com o MIDI e o `Sequencer seqSomDeFundo` para música de fundo.

```
seqSomDeFundo = MidiSystem.getSequencer();
        seqSomDeFundo.setSequence(
                MidiSystem.getSequence(
                    new File("som/piano_quebrado.mid")));

        seqSomDeFundo.open();
        seqSomDeFundo.setLoopCount(Sequencer.LOOP_CONTINUOUSLY);
        seqSomDeFundo.start();

    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Ao contrário dos efeitos que acontecerão em determinados momentos do jogo, nossa música de fundo é iniciada no carregamento e tocada de forma continua.

Dica

Para saber quais formatos seu S.O. suporta:

```
Type[] audioFileTypes = AudioSystem.getAudioFileTypes();
for (Type t : audioFileTypes) {
    System.out.println(t.getExtension());
}
```

Como não estamos tocando os clipes de áudio em loop, temos de reiniciá-los (`setFramePosition(0)`) quando eles chegam ao fim. Assim, reiniciamos primeiro e tocamos depois.

```
...
} else if (temporizador >= 20) {
    temporizador = 0;

    if (colidiu(ppx, ppy + 1)) {

        if (clipAdicionarPeca != null) {
            clipAdicionarPeca.setFramePosition(0);
            clipAdicionarPeca.start();
        }

...
} else
    temporizador += nivel;
...
```

Na atualização do jogo, quando verificarmos que a peça colidiu, iniciaremos nosso efeito sonoro. De forma parecida, tocaremos nosso efeito para completar linhas.

```
private void descerColunas() {
    ...
    if (clipMarcarLinha != null) {
        clipMarcarLinha.setFramePosition(0);
        clipMarcarLinha.start();
    }

}
```

Vale notar que, se fosse um objeto `Sequencer` em vez de `Clip`, chamaríamos o método `setTickPosition(0)` no lugar de `setFramePosition(0)`.

Por fim, antes de sair do `JogoCenario`, não esqueça de desligar o som.

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

```
    if (clipAdicionarPeca != null) {
        clipAdicionarPeca.stop();
        clipAdicionarPeca.close();
    }

    if (clipMarcarLinha != null) {
        clipMarcarLinha.stop();
        clipMarcarLinha.close();
    }

    if (seqSomDeFundo != null) {
        seqSomDeFundo.stop();
        seqSomDeFundo.close();
    }
}
```

Agora que estamos trabalhando com recursos externos, nosso jogo pode não iniciar tão rapidamente como antes. Então, na classe `Jogo.java`, dentro do método `iniciarJogo`, depois que o jogador escolher em que fase deseja começar, adicionamos uma mensagem para avisar do carregamento do próximo cenário.

```
if (cenario instanceof InicioCenario) {
    cenario.descarregar();
    cenario = null;
    cenario = new JogoCenario(tela.getWidth(), tela.getHeight());

    g2d.setColor(Color.WHITE);
    g2d.drawString("Carregando...", 20, 20);
    tela.repaint();

    cenario.carregar();
...
```

Se estiver em dúvida sobre utilizar MIDI ou WAV, leve em consideração que o MIDI é fácil de criar e menor, mas a qualidade sonora depende da placa de som, então pode haver diferenças entre diferentes computadores. Já o WAV, sem compactação, é um arquivo muito maior, mas tem mais

qualidade de som e tende a ser o mesmo em qualquer computador.

Enquanto desenvolvia, percebi que, mesmo o WAV sendo maior, ele carregou mais rápido e não teve atraso entre a chamada e a execução. Por isso, utilizei MIDI como som de fundo e WAV para efeitos sonoros. Em todo caso, ambos os formatos para os áudios `adiciona_peca` e `piano_quebrado` estão no projeto, para que você possa fazer suas próprias comparações e decidir o que fica melhor no seu jogo (http://www.abyssmedia.com/midirenderer/midi-vs-wav.shtml) .

5.6 Programar, desenhar e ainda ter de compor?

Temos muita coisa pronta na internet, e músicas e efeitos sonoros podem ser encontrados em:

- http://freesound.org;
- http://soundbible.com;
- http://www.soundjay.com;
- http://pt.audiomicro.com (pago).

Mas é bem provável que seus arquivos precisem de alguns ajustes, ou que você os encontre em um formato e queira convertê-los em outro. Para esses e outros ajustes, recomendo:

- *Audacity* – Editor de áudio digital livre (http://audacity.sourceforge.net) ;
- *Conversion-tool.com* – Conversor online que vai além dos formatos de áudio (http://www.conversion-tool.com) .

O arquivo `109662_grunz_success.wav`, usado na hora de marcar linha, é um efeito sonoro baixado no Freesound e editado no Audacity.

5.7 Codificando o jogo

Realmente não imagino como este jogo seria construído sem o uso de arrays, e fizemos código suficiente para percorrê-los de quase todas as formas possíveis. Tudo está disponível em https://github.com/logicadojogo/fontes/tree/master/Cap05.

Exageros à parte, temos uma classe nova chamada `Peca.java`, que ficou responsável pela construção das peças do jogo, ao contrário da nossa classe `JogoCenario.java`, onde a complexa lógica do jogo foi dividida em métodos menores e específicos.

Dentro da pasta som, você encontra os seguintes arquivos: `109662_grunz_success.wav`, `adiciona_peca.mid`, `adiciona_peca.wav`, `piano_quebrado.mid` e `piano_quebrado.wav`.

Algumas modificações na classe `Jogo.java`, `InicioCenario.java` e só.

Não fizemos aqui

- Ter a opção de jogar com e sem som, ou somente efeitos sonoros;
- Manipular a música de fundo durante o jogo.

Melhore você mesmo

- Adicionar um botão para girar a peça no sentido horário;
- Colocar sua própria música de fundo e efeitos sonoros.

5.8 Resumo

Nosso jogo de 7 peças de 4 quadrados tem uma lógica complexa e, em nenhum outro jogo deste livro, a palavra array será usada tantas vezes.

Começamos pelas peças que são simples de serem construídas. Depois, com base no tamanho da grade e da tela, definimos a largura e altura em que

as peças serão desenhadas. Mesmo elas sendo adicionadas aleatoriamente, conseguimos saber qual virá na sequência. Depois, vimos como posicioná-las, preenchendo e desenhando a grade.

Durante o jogo, validamos a movimentação, a colisão e a rotação da peça, e temos duas versões diferentes para girá-las: uma que reposiciona e outra que não. Se ela parar fora da grade, ultrapassando o topo da tela, o jogador perde. Mas após 9 níveis e 81 linhas feitas, ele ganha o jogo; embora o jogador possa escolher em qual level começar, ele estaria abrindo mão de uma pontuação maior. Ainda animamos o jogo e criamos algumas formas de depuração.

Com o jogo pronto, melhoramos com efeitos sonoros, usando as classes nativas do Java. Com certeza, a maioria dos jogos no estilo quebra-cabeças (*puzzle*) que você desenvolver será mais fácil depois deste capítulo.

Capítulo 6

Pac-Man, vulgo Come-come

Criado por Toru Iwatani em 1979, lançado em 1980, Pac-Man queria se diferenciar dos jogos de tiro da época e atrair o público feminino, mas ele conseguiu bem mais que isso, tornou-se um símbolo internacional dos jogos de videogames, um superastro inspirado em um pedaço de pizza.

Desenvolver um jogo como o Pac-Man sem usar imagens é, no mínimo, ofensivo, mas ele envolve muitos conceitos complexos ainda não abordados. Então, seguindo o ditado "dividir para conquistar", neste capítulo abordaremos aspectos relacionados à lógica do jogo, e depois veremos as principais formas de trabalharmos com imagens, além de:

- Perseguição e fuga com Inteligência Artificial;
- Fluxo de estados dos personagens;
- Colisão de elementos em eixo com itens no array.

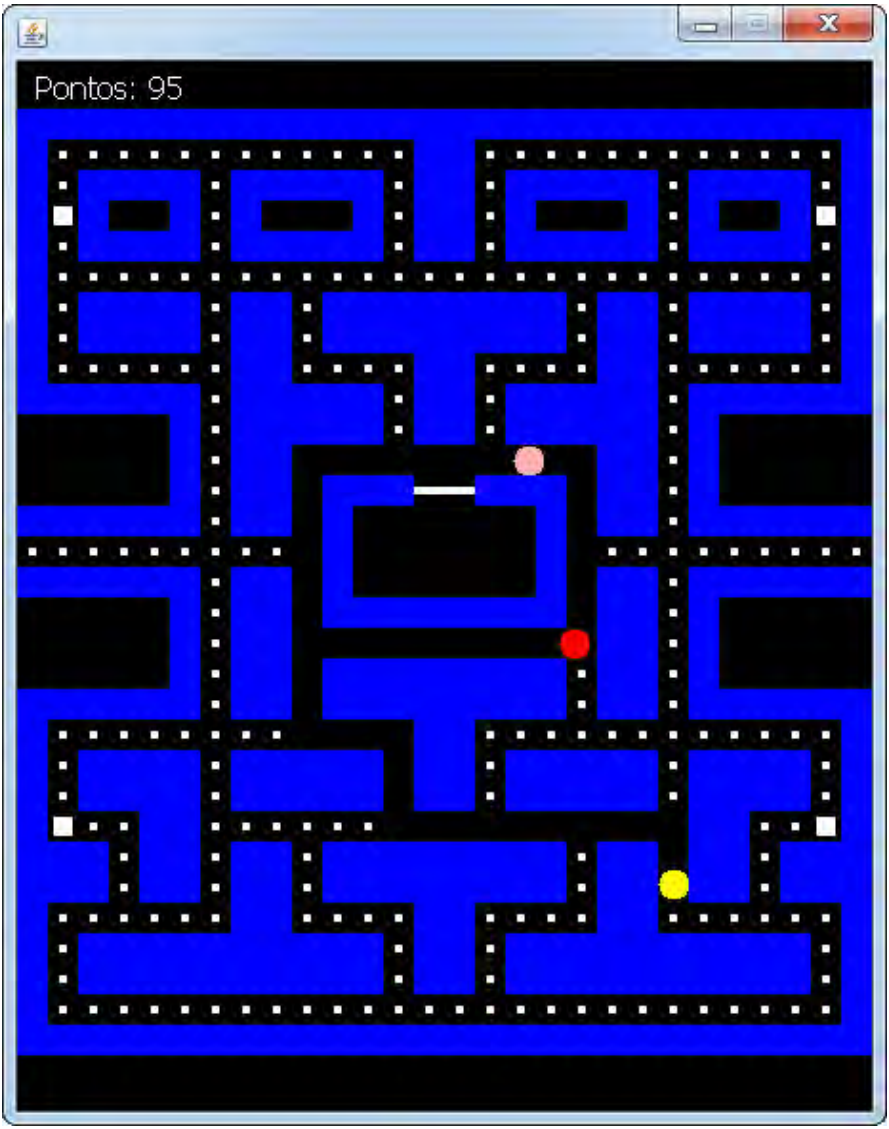

Fig. 6.1: Come-come

6.1 Caça fantasma

Já vimos como montar cenários e brincamos bastante com array. A novidade aqui é que, embora nosso cenário seja um array (formando uma grade como no Tetris), os elementos se movem pixel a pixel (parecido com nosso jogo da cobrinha), e não em linhas e colunas, tendo assim uma movimentação fluida.

Montamos alguns níveis no capítulo 4, então você já sabe como isso será feito aqui. Este é nosso cenário:

```
public static int[][] cenario = {
    { BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL },
    { BL, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, BL, BL, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, BL },
    { BL, CN, BL, BL, BL, BL, CN, BL, BL, BL, BL, BL, CN, BL, BL, CN, BL, BL, BL, BL, CN, BL, BL, BL, BL, CN, BL },
    { BL, SC, BL, EV, EV, BL, CN, BL, EV, EV, EV, BL, CN, BL, BL, CN, BL, EV, EV, EV, BL, CN, BL, EV, EV, BL, SC, BL },
    { BL, CN, BL, BL, BL, BL, CN, BL, BL, BL, BL, BL, CN, BL, BL, CN, BL, BL, BL, BL, BL, CN, BL, BL, BL, BL, CN, BL },
    { BL, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, BL },
    { BL, CN, BL, BL, BL, BL, CN, BL, BL, CN, BL, BL, BL, BL, BL, BL, BL, CN, BL, BL, CN, BL, BL, BL, BL, CN, BL },
    { BL, CN, BL, BL, BL, BL, CN, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, CN, BL, BL, CN, BL, BL, BL, BL, CN, BL },
    { BL, CN, CN, CN, CN, CN, CN, BL, BL, CN, CN, CN, CN, BL, BL, CN, CN, CN, CN, BL, BL, CN, CN, CN, CN, CN, BL },
    { BL, BL, BL, BL, BL, BL, CN, BL, BL, BL, BL, BL, CN, BL, BL, CN, BL, BL, BL, BL, CN, BL, BL, BL, BL, BL, BL },
    { EV, EV, EV, EV, EV, BL, CN, BL, BL, BL, BL, BL, CN, BL, BL, CN, BL, BL, BL, BL, BL, CN, BL, EV, EV, EV, EV, EV },
    { EV, EV, EV, EV, EV, BL, CN, BL, BL, EV, EV, EV, PF, P1, EV, EV, EV, EV, BL, BL, CN, BL, EV, EV, EV, EV, EV },
    { EV, EV, EV, EV, EV, BL, CN, BL, BL, EV, BL, BL, BL, LN, LN, BL, BL, BL, EV, BL, BL, CN, BL, EV, EV, EV, EV, EV },
    { BL, BL, BL, BL, BL, BL, CN, BL, BL, EV, BL, EV, EV, EV, EV, EV, EV, BL, EV, BL, BL, CN, BL, BL, BL, BL, BL, BL },
    { CN, CN, CN, CN, CN, CN, CN, CN, CN, EV, BL, EV, EV, EV, EV, EV, EV, BL, EV, CN, CN, CN, CN, CN, CN, CN, CN, CN },
    { BL, BL, BL, BL, BL, BL, CN, BL, BL, EV, BL, EV, P2, PV, EV, EV, EV, BL, EV, BL, BL, CN, BL, BL, BL, BL, BL, BL },
    { EV, EV, EV, EV, EV, BL, CN, BL, BL, EV, BL, BL, BL, BL, BL, BL, BL, EV, BL, BL, CN, BL, EV, EV, EV, EV, EV, EV },
    { EV, EV, EV, EV, EV, BL, CN, BL, BL, EV, EV, EV, EV, PI, EV, EV, EV, EV, EV, BL, BL, CN, BL, EV, EV, EV, EV, EV },
    { EV, EV, EV, EV, EV, BL, CN, BL, BL, CN, BL, BL, BL, BL, BL, BL, BL, CN, BL, BL, CN, BL, EV, EV, EV, EV, EV, EV },
    { BL, BL, BL, BL, BL, BL, CN, BL, BL, CN, BL, BL, BL, BL, BL, BL, BL, CN, BL, BL, CN, BL, BL, BL, BL, BL, BL, BL },
    { BL, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, BL, BL, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, BL },
    { BL, CN, BL, BL, BL, BL, CN, BL, BL, BL, BL, BL, CN, BL, BL, CN, BL, BL, BL, BL, CN, BL, BL, BL, BL, CN, BL },
    { BL, CN, BL, BL, BL, BL, CN, BL, BL, BL, BL, BL, CN, BL, BL, CN, BL, BL, BL, BL, CN, BL, BL, BL, BL, CN, BL },
    { BL, SC, CN, CN, BL, BL, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, BL, BL, CN, CN, SC, BL },
    { BL, BL, BL, CN, BL, BL, CN, BL, BL, CN, BL, BL, BL, BL, BL, BL, BL, CN, BL, BL, CN, BL, BL, CN, BL, BL, BL },
    { BL, BL, BL, CN, BL, BL, CN, BL, BL, CN, BL, BL, BL, BL, BL, BL, BL, CN, BL, BL, CN, BL, BL, CN, BL, BL, BL },
    { BL, CN, CN, CN, CN, CN, CN, BL, BL, CN, CN, CN, CN, BL, BL, CN, CN, CN, CN, BL, BL, CN, CN, CN, CN, CN, BL },
    { BL, CN, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, CN, BL, BL, CN, BL, BL, BL, BL, BL, BL, BL, BL, BL, CN, BL },
    { BL, CN, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, CN, BL, BL, CN, BL, BL, BL, BL, BL, BL, BL, BL, BL, CN, BL },
    { BL, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, BL },
    { BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL, BL } };
```

Fig. 6.2: Cenário

Em vez de utilizarmos valores inteiros diretamente, vamos separá-los em constantes:

```
public static final int BL = 0; /* Bloco */
public static final int CN = 1; /* Comida normal */
public static final int EV = 2; /* Espaco vazio */
public static final int PI = 3; /* Ponto inicial do jogador */
public static final int LN = 4; /* Linha */
public static final int SC = 5; /* Super comida */
public static final int P1 = 6; /* Ponto inicial inimigo 1 */
public static final int P2 = 7; /* Ponto inicial inimigo 2 */
```

```
public static final int PF = 8; /* Ponto de Fuga */
public static final int PV = 9; /* Ponto de Volta */
```

Como estamos utilizando diversos valores, armazená-los em constantes facilitará a programação do jogo. Alguns valores são usados para montar nosso labirinto (0 a 5), outros servem para auxiliar na movimentação e configuração dos personagens (6 a 9). Começando por:

- `BL` (*BL*oco), usado para formar a parede do nosso labirinto.
- `CN` (*C*omida *N*ormal), que representa as pastilhas/pílulas que devem ser devoradas.
- `SC` (*S*uper*C*omida) é a pílula maior que dá poder ao personagem.
- `EV` (*E*spaço *V*azio) é a parte sem item do cenário.
- Uma pastilha ou supercomida que, após ser "comida", vira um espaço vazio.
- `LN` (*Li*N*h*a) é a parede especial por onde somente os inimigos podem passar.

Para configurar o jogo, temos:

- `PI` (*P*onto *I*nicial) usado para determinar onde nosso herói começará.
- `P1` e `P2`, que são as posições iniciais de cada inimigo (nessa versão do jogo, só teremos dois).
- `PF` (*P*onto de Fuga) e `PV` (*P*onto de Volta), que têm funções opostas, o primeiro indica onde é a saída para o inimigo que está preso (fica do lado de fora da prisão); o segundo serve para que o inimigo que virou fantasma seja ressuscitado (fica do lado de dentro da prisão).

Assim, poderemos montar algo parecido com o jogo original:

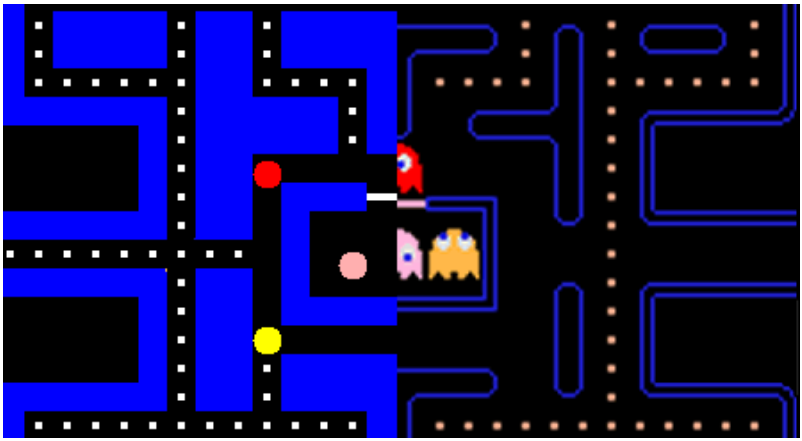

Fig. 6.3: Baseado em cenários reais

Agora que vimos a grade e tudo o que podemos colocar nela, veremos como ela é usada no jogo. Temos aqui um array 31x28, isso é, um array contendo 31 posições e cada uma dessas posições tem outro array com 28 posições. O primeiro usado para representar as linhas (`grade.length`), o segundo, as colunas (`grade[0].length`).

Cada item na tela terá o tamanho de 16px, resultando na altura de 496px (31*16) e largura de 448px (28*16). Mas, para ficar mais parecido com o jogo original, aumentamos a altura da tela para 550px, deixando espaço acima e abaixo do labirinto. Armazenamos os valores desse espaçamento na constante estática `JogoCenario.ESPACO_TOPO`.

Nossos círculos coloridos são os únicos objetos que se movem, então eles não são simples `Elementos`, são elementos `Pizza`.

```
public class Pizza extends Elemento {

    public enum Modo {
        PRESO, ATIVO, INATIVO, FANTASMA, CACANDO, FUGINDO;
    }

    private int dx, dy;
    private Modo modo = Modo.PRESO;
```

```
    private Direcao direcao = Direcao.OESTE;

    public Pizza(int px, int py, int largura, int altura) {
        super(px, py, largura, altura);
    }

    @Override
    public void desenha(Graphics2D g) {
        if (modo == Modo.FUGINDO)
            g.setColor(Color.LIGHT_GRAY);
        else
            g.setColor(getCor());

        if (modo == Modo.FANTASMA)
           g.drawOval(getPx(), getPy() + JogoCenario.ESPACO_TOPO,
                  getLargura(), getAltura());
        else
           g.fillOval(getPx(), getPy() + JogoCenario.ESPACO_TOPO,
                  getLargura(), getAltura());
    }
```

Embora apenas os inimigos tenham modos/estados diferenciados, eles são muito utilizados durante o jogo, inclusive para determinar como serão desenhados:

- Círculo preenchido na cor cinza quando estão fugindo (`Modo.FUGINDO`);
- Círculo sem preenchimento caso tenham virado fantasma (`Modo.FANTASMA`);
- Círculo preenchido com a cor individual pré-definida para os outros estados.

Seja qual for o formato do desenho, levamos em conta o espaçamento superior somando-o ao eixo Y.

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

```
        incPx(getVel() * getDx());
        incPy(getVel() * getDy());
    }

...
}
```

Outra novidade é que controlamos a direção dos personagens usando pontos cardeais: `NORTE`, `SUL`, `OESTE` e `LESTE`. Como a direção é controla pelo jogador, nosso `enum Direcao` fica na classe `JogoCenario.java`.

Nossa variável `direcao` é mais usada para indicar para onde o jogador pretende ir, não necessariamente para onde o elemento está indo, deixando o controle real da movimentação a cargo de `dx` e `dy`, como já visto em outros capítulos.

Falaremos mais sobre a movimentação dos personagens, mas antes vamos ver como eles são criados dentro do método `carregar` da classe `JogoCenario`:

```
...
largEl = largura / grade[0].length; // 16px

pizza = new Pizza(0, 0, largEl, largEl);
pizza.setVel(4);
pizza.setAtivo(true);
pizza.setCor(Color.YELLOW);
pizza.setDirecao(Direcao.OESTE);

// Inimigos
inimigos = new Pizza[2];

inimigos[0] = new Pizza(0, 0, largEl, largEl);
inimigos[0].setVel(3 + Jogo.nivel);
inimigos[0].setAtivo(true);
inimigos[0].setCor(Color.RED);
inimigos[0].setDirecao(Direcao.OESTE);
inimigos[0].setModo(Pizza.Modo.CACANDO);
```

Primeiro, achamos o espaço disponível para nossos elementos na grade e

guardamos na variável `largEl`. Neste caso, dividimos a largura da tela pela quantidade de colunas, 448 dividido por 28, resultando em 16 pixels.

Embora a altura da tela seja maior (550px), a grade ocupa menos espaço (496), que dividido por 31 linhas, também resulta em 16 pixels. Por isso, usamos esse mesmo valor tanto para largura quanto altura dos personagens.

Nosso personagem principal, carinhosamente chamado de `pizza`, começa ativo, na cor amarela, movendo-se para a esquerda (Oeste). Para controlar os antagonistas, usamos um array, chamado `inimigos`. A velocidade deles varia de acordo com a dificuldade escolhida pelo jogador.

```
inimigos[1] = new Pizza(0, 0, largEl, largEl);
inimigos[1].setVel(2 + Jogo.nivel);
inimigos[1].setAtivo(false);
inimigos[1].setCor(Color.PINK);
inimigos[1].setDirecao(Direcao.NORTE);
inimigos[1].setModo(Pizza.Modo.PRESO);
```

Embora o primeiro inimigo, na cor vermelha, comece ativo e caçando nosso jogador pela tela, o segundo, na cor rosa, começa inativo, preso e se move mais lentamente. Todos são iniciados na posição 0, já que precisaremos percorrer a grade para obter as posições iniciais de cada um:

```
for (int lin = 0; lin < grade.length; lin++) {
    for (int col = 0; col < grade[0].length; col++) {
        if (grade[lin][col] == Nivel.CN ||
            grade[lin][col] == Nivel.SC) {
            totalPastilha++;

        } else if (grade[lin][col] == Nivel.PI) {
            pizza.setPx(converteInidicePosicao(col));
            pizza.setPy(converteInidicePosicao(lin));

        } else if (grade[lin][col] == Nivel.P1) {
            inimigos[0].setPx(converteInidicePosicao(col));
            inimigos[0].setPy(converteInidicePosicao(lin));

        } else if (grade[lin][col] == Nivel.P2) {
            inimigos[1].setPx(converteInidicePosicao(col));
```

```
                inimigos[1].setPy(converteInidicePosicao(lin));

            } else if (grade[lin][col] == Nivel.PF) {
                pontoFugaCol = col;
                pontoFugaLin = lin;

            } else if (grade[lin][col] == Nivel.PV) {
                pontoVoltaCol = col;
                pontoVoltaLin = lin;
            }
        }
}
```

Aproveitamos também para somar a quantidade total de pastilhas que nosso jogador terá de comer para ganhar o jogo, além de guardarmos a linha e coluna do ponto de fuga e ponto de volta, que servem de bússola para nossos inimigos saírem e voltarem para prisão. Observe que utilizamos a função `converteInidicePosicao`, isso para suavizar a movimentação dos nossos personagens, pois eles movem-se nos eixos alguns pixels por vez, e não em linha/coluna, como no Tetris.

Isso seria mais simples se os outros elementos do jogo (como pastilhas e blocos) não estivessem fixos em suas linhas e colunas. Logo, para isso funcionar corretamente, temos também a função `convertePosicaoIndice`, mas não se preocupe, elas realmente não possuem nenhum mistério:

```
private int converteInidicePosicao(int linhaColuna) {
    return linhaColuna * largEl;
}

private int convertePosicaoIndice(int eixoXY) {
    return eixoXY / largEl;
}
```

A primeira converte um índice, posição da linha ou coluna do array, em eixo X ou Y. A segunda, o inverso, um eixo X ou Y em índice. Vale lembrar que X equivale à coluna, e Y à linha. Exemplificando, se quisermos saber qual a posição X do personagem na coluna 21, multiplicamos 21 pelo tamanho que

cada espaço da grade ocupa na tela (16), então X será 336. O mesmo para a linha e o eixo Y.

Agora, se estivermos no eixo Y 250, a linha equivalente seria 250 dividido por 16, neste caso 15. Note que esse resultado seria 15,625 se estivéssemos trabalhando com `float` em vez de `int`, e 16 caso arredondássemos com `Math.round`, mas este não é o caso. Esperamos que o valor seja realmente 15, nem mais, nem menos.

Dito isso, podemos `atualizar` o jogo, lembrando de que nossos personagens só podem ir em uma das quatro direções enquanto se movem.

```
if (Jogo.controleTecla[Jogo.Tecla.ESQUERDA.ordinal()]) {
    prxDirecao = Direcao.OESTE;
} else if (Jogo.controleTecla[Jogo.Tecla.DIREITA.ordinal()]) {
    prxDirecao = Direcao.LESTE;
} else if (Jogo.controleTecla[Jogo.Tecla.CIMA.ordinal()]) {
    prxDirecao = Direcao.NORTE;
} else if (Jogo.controleTecla[Jogo.Tecla.BAIXO.ordinal()]) {
    prxDirecao = Direcao.SUL;
}
pizza.setDirecao(prxDirecao);
```

Guardamos a direção escolhida pelo jogador na variável `prxDirecao`, convertendo a tecla pressionada em ponto cardeal. Isso é feito para ajudar na jogabilidade (*gameplay*), não obrigando o jogador a acertar o momento exato em que o personagem poderá ir na direção escolhida. Por exemplo, se o personagem estiver se movendo na horizontal e o jogador pressionar para cima, caso o personagem possa ir nesta direção não haverá nenhum problema, caso contrário, já sabemos para onde o jogador quer ir e tentaremos novamente até termos uma movimentação válida ou o jogador escolher outra direção. Assim não deixamos o jogador executar um movimento inválido sem obrigá-lo a executar o movimento no momento exato.

Isso fica a cargo do método `atualizarDirecao`, que verifica se o jogador realmente pode ir na direção escolhida.

Ainda precisamos corrigir sua posição – já que ao sair da tela pelo lado esquerdo os personagens aparecem no lado direito e vice-versa –, verificar se personagem andou se alimentando e, só então, chamar o método `atualizar`.

```
atualizaDirecao(pizza);
corrigePosicao(pizza);
comePastilha(pizza);
pizza.atualiza();

if (superPizza && temporizadorPizza > 200) {
    temporizadorPizza = 0;
    superPizza(false);
} else
    temporizadorPizza += 1;
```

Se o jogador comer a pastilha especial, ele se transforma no superpizza, então usamos um temporizador para controlar o tempo em que o jogador ficará invulnerável.

Chamamos `superPizza(true)` para transformá-lo, e `superPizza(false)` para voltá-lo ao normal. Depois de tudo isso, passamos para os inimigos.

```
for (Pizza el : inimigos) {
    if (el == null)
        continue;

    atualizaDirecaoInimigos(el);
    corrigePosicao(el);
    el.atualiza();

    if (Util.colide(pizza, el)) {

        if (el.getModo() == Pizza.Modo.CACANDO) {
            reiniciar(); // Jogador perdeu
        } else if (el.getModo() == Pizza.Modo.FUGINDO) {
            el.setAtivo(false);
            el.setModo(Pizza.Modo.FANTASMA);
            pontos += 50;
        }
    }
}
```

O método que atualiza a direção do jogador é diferente do que atualiza

a direção dos inimigos, afinal, o jogador só precisa indicar para onde quer ir. Entretanto, os inimigos precisam de um pouco de Inteligência Artificial, como veremos mais à frente. Por fim, quando ocorre uma colisão do inimigo com o jogador, verificamos se o inimigo estava caçando o jogador (ele entra nesse modo quando atinge o ponto de fuga), ou se estava fugindo dele – nesse caso, o inimigo estava caçando o jogador quando este virou superpizza.

Quando o inimigo está caçando, a colisão não resulta na perda do jogo, todos os personagens apenas voltam para suas posições iniciais. Agora, se o inimigo estava fugindo, ele vira um fantasma que, em vez de perseguir o jogador, busca o ponto de volta e o jogador ganha mais pontos.

Quando atualizamos a direção dos nossos personagens, precisamos validar a direção e também a movimentação deles. Então veremos esses métodos primeiro, começando pelo `validaDirecao`, que apenas chama o método `validaMovimento` com os valores correspondentes à direção escolhida.

```
private boolean validaDirecao(Direcao dir, Pizza el) {

    if (dir == Direcao.OESTE && validaMovimento(el, -1, 0))
        return true;

    else if (dir == Direcao.LESTE && validaMovimento(el, 1, 0))
        return true;

    else if (dir == Direcao.NORTE && validaMovimento(el, 0, -1))
        return true;

    else if (dir == Direcao.SUL && validaMovimento(el, 0, 1))
        return true;

    return false;
}
```

Sabendo para qual direção o jogador quer ir, precisamos transformar essa direção na movimentação propriamente dita, por exemplo, Oeste equivale à esquerda, que equivale a subtrair a posição do elemento no eixo X.

Então, considerando que a direção escolhida foi Oeste, estamos validando o movimento utilizando -1 em X, o para Y e a velocidade para saber quantos

pixels o personagem vai se mover (na direção que for diferente de zero).

Seguindo a mesma lógica para as demais direções, embora somando 1 no lugar de subtrair para direção Leste e, para Norte e Sul, subtraindo e somando 1 em Y respectivamente, deixando 0 em X.

Se tudo que vimos até agora parecia simples, nosso próximo método destoa do restante do código, pelo menos à primeira vista. Para validar o movimento, precisamos detectar a colisão dos nossos elementos com os valores na grade e, assim como a detecção de colisão entre elementos, precisamos levar em conta a posição e o tamanho deles.

```
private boolean validaMovimento(Pizza el, int dx, int dy) {
    // Próxima posição x e y
    int prxPosX = el.getPx() + el.getVel() * dx;
    int prxPosY = el.getPy() + el.getVel() * dy;

    // Coluna e linha
    int col = convertePosicaoIndice(prxPosX);
    int lin = convertePosicaoIndice(prxPosY);

    // Coluna + largura e linha + altura
    int colLarg = convertePosicaoIndice(prxPosX +
                    el.getLargura() - el.getVel());
    int linAlt = convertePosicaoIndice(prxPosY +
                        el.getAltura() - el.getVel());
```

Estamos validando o movimento que ainda não ocorreu, então precisamos simular a posição futura do elemento. Para isso, é necessário ter a posição do elemento mais a velocidade (quantidade de pixels a serem incrementados) multiplicada pela direção (-1, 0 ou 1).

Convertemos a posição futura em coluna e linha para saber a posição do elemento na grade, mas isso não é o suficiente. Quando o personagem se move para esquerda ou para baixo, precisamos levar em conta seu tamanho para acharmos a coluna e linha corretas. Logo, somamos a próxima posição no eixo X à largura do elemento, subtraindo a velocidade, que já está embutida em `prxPosX`. Assim, acharemos a coluna correta do elemento quando ele estiver se movendo para direita, o equivalente para o `prxPosY` e altura do elemento se movendo para baixo.

```
if (foraDaGrade(col, lin) || foraDaGrade(colLarg, linAlt))
    return true;

if (grade[lin][col] == Nivel.BL ||
    grade[lin][colLarg] == Nivel.BL ||
    grade[linAlt][col] == Nivel.BL ||
    grade[linAlt][colLarg] == Nivel.BL) {
    return false;
}
```

Como os personagens podem atravessar de um lado para o outro do labirinto ao saírem da tela, e nesse momento eles estarão fora do array, usamos a função `foraDaGrade` para verificar se isso ocorreu – neste caso, retornamos verdadeiro, interrompendo a validação para que eles não parem no meio do caminho.

Se ele não estiver fora da grade, verificamos se o personagem colidiu com algum bloco, seja indo para cima e para esquerda, para cima e para direita, para baixo e para esquerda, ou para baixo e para direita.

O que chamamos de linha (`Nivel.LN`) é como se fosse um bloco especial onde os inimigos só podem atravessar se estiverem inativos ou não estiverem presos (`Modo.PRESO`).

```
// Validar linha branca
    if (el.isAtivo() || el.getModo() == Modo.PRESO) {
        if (grade[lin][col] == Nivel.LN ||
            grade[lin][colLarg] == Nivel.LN ||
            grade[linAlt][col] == Nivel.LN ||
            grade[linAlt][colLarg] == Nivel.LN) {
            return false;
        }
    }

    return true;
}
```

Apenas repetimos a mesma verificação, trocando `Nivel.BL` por `Nivel.LN`, para que os inimigos ativos e até mesmo o jogador não entrem acidentalmente na jaula. Se não houver colisão, retornamos `true`, indicando

que o movimento é válido. Com esses métodos em mente, chegou a hora de vermos como o jogo funciona.

6.2 Jogando o código

Nosso jogo tem três personagens: nosso herói representado por um círculo amarelo e dois inimigos, um vermelho e um rosa. Cada personagem tem sua posição inicial definida no próprio nível, mesmo assim os deixamos nas posições semelhantes ao jogo original. O inimigo vermelho começa fora da jaula, já o inimigo rosa começa preso e demora um pouco para se libertar.

Para nosso herói ganhar o jogo, ele tem de comer todas as pastilhas da tela. Porém, não adicionamos um fim de jogo. Ao ser pego por um fantasma, apenas reiniciamos as posição do nosso herói e de seus inimigos.

Esses são apenas os aspectos básicos do jogo, que qualquer jogador consegue perceber. O que fica por baixo dos panos, que nós, desenvolvedores, conseguimos ver, é bem mais interessante, como por exemplo, o fluxo de atividades que nossos inimigos seguem:

Fig. 6.4: Fluxo fantasma

Em cada um desses seis estados que nossos inimigos podem ter, eles estarão preocupados com coisas diferentes, então cada estado tem um toque de inteligência, mesmo que seja artificial. Por isso, teremos uma seção própria só para falarmos disso, mas antes, veremos mais aspectos relacionados ao nosso herói.

Pode ocorrer da velocidade que você definir para algum personagem acabar posicionando-o em um local inadequado da tela, com uma distância perceptível da parede ou um pouco dentro dela, por exemplo. Para evitar isso e garantir que nossos personagens possam sair da área visível da tela (sendo posicionados do lado oposto), utilizamos uma única função.

```
private void corrigePosicao(Pizza el) {
    int novaPx = el.getPx(); // Nova posição x
    int novaPy = el.getPy(); // Nova posição y
```

```
    int col = convertePosicaoIndice(el.getPx()) * largEl;
    int lin = convertePosicaoIndice(el.getPy()) * largEl;

    if (el.getDx() == 0 && novaPx != col)
        novaPx = col;
    else if (el.getPx() + largEl < 0)
        novaPx = largura;
    else if (el.getPx() > largura)
        novaPx = -largEl;
```

Eu chamaria essa função de teleporte sem problemas, mas a primeira verificação é para sabermos se o elemento não está se movendo na vertical (eixo X). Assim, o resultado da validação de colisão pode tê-lo posicionado fora dos 16px de espaçamento de cada coluna e, se isso acontecer, definimos a nova posição com o valor da posição da coluna mais próxima para enquadrá-lo na grade.

Depois, verificamos se o personagem saiu totalmente da tela, fazendo os personagens se teletransportarem de um lado para o outro. Embora nosso labirinto não tenha saídas horizontais, deixamos um método equivalente para o teleporte no eixo Y, caso precise.

```
if (el.getDy() == 0 && novaPy != lin)
        novaPy = lin;
    else if (el.getPy() + largEl < 0)
        novaPy = altura;
    else if (el.getPy() > altura)
        novaPy = -largEl;

    el.setPx(novaPx);
    el.setPy(novaPy);
}
```

Por fim, atualizamos as posições do elemento em seus respectivos eixos. Vale notar que somamos e subtraímos o valor da variável `largEl` por ser equivalente ao tamanho do elemento. Muito parecido com nosso método para verificar se o personagem saiu da tela:

```
private boolean foraDaTela(Elemento el) {
    if (el.getPx() < 0 || el.getPx() + el.getLargura() > largura)
```

```
        return true;

    if (el.getPy() < 0 || el.getPy() + el.getAltura() > altura)
        return true;

    return false;
}
```

E como estamos trabalhando também com a posição do personagem em coluna e linha:

```
private boolean foraDaGrade(int coluna, int linha) {
    if (linha < 0 || linha >= grade.length)
        return true;

    if (coluna < 0 || coluna >= grade[0].length)
        return true;

    return false;
}
```

Dois métodos já conhecidos, vistos em outros capítulos. A principal atividade do nosso personagem é se alimentar e, para isso, precisamos verificar se ele colidiu com alguma pastilha, convertendo suas posições no eixo X e Y em coluna e linha.

```
private void comePastilha(Elemento el) {
    int col = convertePosicaoIndice(el.getPx());
    int lin = convertePosicaoIndice(el.getPy());

    if (foraDaGrade(col, lin)) {
        return;
    }

    if (grade[lin][col] == Nivel.CN ||
            grade[lin][col] == Nivel.SC) {
        pontos += grade[lin][col] == Nivel.CN ? 5 : 25;
        totalPastilha--;
```

```
        if (totalPastilha == 0)
            estado = JogoCenario.Estado.GANHOU;
        else if (grade[lin][col] == Nivel.SC)
            superPizza(true);

        grade[lin][col] = Nivel.EV;
    }
}
```

Se o jogador estiver em uma posição válida na grade e esta coincidir com uma pastilha normal (que vale 5 pontos), ou com uma superpastilha (que vale 25), somamos os pontos e decrementamos o total de pastilhas disponíveis. Se elas acabarem, ele ganha o jogo; se não tiverem acabado ainda e for uma superpastilha, ele entra em modo super. O espaço ocupado pela pastilha é substituído por um espaço vazio.

Nosso herói tem somente dois modos de jogo: normal e super. Ele começa no modo normal e, ao comer uma superpastilha, entra em modo super por alguns segundos e, então, volta ao modo normal quando este tempo acaba – embora essa mudança de normal para super e super para normal não altere o estado do nosso herói.

```
private void superPizza(boolean modoSuper) {
    superPizza = modoSuper;
    temporizadorPizza = 0;

    for (Pizza el : inimigos) {
        if (el == null)
            continue;

        if (modoSuper && el.getModo() == Pizza.Modo.CACANDO)
            el.setModo(Pizza.Modo.FUGINDO);
        else if (!modoSuper && el.getModo() ==
                    Pizza.Modo.FUGINDO)
            el.setModo(Pizza.Modo.CACANDO);
    }
}
```

Usamos o mesmo método para fazer o jogador entrar e sair do modo super. Usamos `superPizza` e `temporizadorPizza` para controlar o tempo

em que o jogador ficará nesse modo. Apenas o estado do inimigo é levado em consideração quando detectarmos colisão entre eles e nosso herói. Isso porque pode haver colisão com os inimigos em diferentes estados (quando eles viram fantasmas, por exemplo) e pode acontecer que mesmo ele estando com o modo superativo, o inimigo já tenha se recuperado e voltado à caça.

Passando verdadeiro, ele entra no modo super e os inimigos ficam vulneráveis; passando falso, ele sai desse modo, ficando vulnerável aos inimigos. Em todo caso, o temporizador é zerado, sendo que somente os inimigos que estão caçando ou fugindo serão afetados por essa mudança.

Já usamos dois inteiros para representar a direção do elemento nos eixos X e Y antes, mas agora, além de `dx` e `dy`, temos também uma variável `direcao`, que representa Norte, Sul, Oeste e Leste. O jogador pode escolher ir para qualquer uma dessas direções, mas nosso personagem só acatará essa decisão no momento certo. Assim, a direção em que o personagem está indo nem sempre será a direção que o jogador escolheu, mas ela não é ignorada. Toda vez que atualizamos o jogo, verificamos se conseguimos atender o desejo do jogador.

```
private void atualizaDirecao(Pizza el) {

    if (foraDaTela(el))
        return;

    // Temporario Direcao X e Y
    int tempDx = el.getDx();
    int tempDy = el.getDy();

    Direcao direcao = el.getDirecao();

    if (validaDirecao(direcao, el)) {
        if (direcao == Direcao.OESTE)
            tempDx = -1;
        else if (direcao == Direcao.LESTE)
            tempDx = 1;

        if (direcao == Direcao.NORTE)
            tempDy = -1;
```

```
        else if (direcao == Direcao.SUL)
            tempDy = 1;

    }

    if (!validaMovimento(el, tempDx, tempDy))
        tempDx = tempDy = 0;

    el.setDx(tempDx);
    el.setDy(tempDy);
}
```

Antes de atualizarmos a direção do jogador, verificamos se o personagem não está fora da tela, o que quer dizer que ele entrou no túnel e aparecerá do seu lado oposto. Assim, ele não terá a chance de fugir do labirinto, quebrando o jogo. Só podemos trocar a direção do personagem se ela for válida, e essa verificação é o papel do `validaDirecao`.

Tendo uma direção válida, atualizamos os valores de `tempDx` e `tempDy`, que serão realmente os valores utilizados na movimentação dos personagens. Por isso, independentemente de ter uma direção válida ou não, precisamos validar seu movimento (`validaMovimento`). Se não tivermos um movimento válido, zeramos ambas as variáveis, fazendo o personagem parar de se mover até que o jogador indique uma nova direção válida.

O método `validaDirecao` utiliza internamente o método `validaMovimento`, apenas convertendo ponto cardeal em eixo. Embora à primeira vista essas validações pareçam complexas, basicamente colocamos os personagens na posição futura e verificamos se não acabaram dentro de uma parede. Vale lembrar que nossos personagens só podem se mover na horizontal (Oeste, Leste ou eixo X) ou na vertical (Norte, Sul ou eixo Y).

Já mencionamos que, quando nosso jogador é pego por um inimigo, reiniciamos seus estados e suas posições.

```
public void reiniciar() {
    superPizza = false;
    temporizadorFantasma = 0;
    prxDirecao = Direcao.OESTE;
```

```
    pizza.setDirecao(Direcao.OESTE);

    inimigos[0].setDirecao(Direcao.OESTE);
    inimigos[0].setModo(Pizza.Modo.CACANDO);
    inimigos[0].setAtivo(true);

    inimigos[1].setDirecao(Direcao.NORTE);
    inimigos[1].setModo(Pizza.Modo.PRESO);
    inimigos[1].setAtivo(false);
```

Desabilitamos o modo super do nosso herói e deixamos todos como se tivéssemos acabado de carregar o jogo. Depois, achamos novamente suas posições iniciais na grade.

```
for (int lin = 0; lin < grade.length; lin++) {
        for (int col = 0; col < grade[0].length; col++) {
            if (grade[lin][col] == Nivel.PI) {
                pizza.setPx(converteInidicePosicao(col));
                pizza.setPy(converteInidicePosicao(lin));

            } else if (grade[lin][col] == Nivel.P1) {
                inimigos[0].setPx(converteInidicePosicao(col));
                inimigos[0].setPy(converteInidicePosicao(lin));

            } else if (grade[lin][col] == Nivel.P2) {
                inimigos[1].setPx(converteInidicePosicao(col));
                inimigos[1].setPy(converteInidicePosicao(lin));
            }
        }
    }
}
```

Para evitar percorrer todo o array novamente, poderíamos ter guardado as posições dos personagens em cada elemento, mas não há nenhuma novidade aqui. Passamos, então, para o método `desenhar`, que desenha tanto os valores da grade quanto nossos objetos `Pizza`.

```
for (int lin = 0; lin < grade.length; lin++) {
    for (int col = 0; col < grade[0].length; col++) {
```

```
            int valor = grade[lin][col];

            if (valor == Nivel.BL) {
                g.setColor(superPizza ? Color.DARK_GRAY :
                            Color.BLUE);
                g.fillRect(col * largEl, lin * largEl + ESPACO_TOPO,
                      largEl, largEl);

            } else if (valor == Nivel.CN) {
                g.setColor(Color.WHITE);
                g.fillRect(col * largEl + espLinha,
                      lin * largEl + espLinha + ESPACO_TOPO,
                      largEl - espLinha * 2,
                      largEl - espLinha * 2 );

            } else if (valor == Nivel.SC) {
                g.setColor(Color.WHITE);
                g.fillRect(col * largEl + espLinha / 2,
                      lin * largEl + espLinha / 2 + ESPACO_TOPO,
                      largEl - espLinha,
                      largEl - espLinha);

            } else if (valor == Nivel.LN) {
                g.setColor(Color.WHITE);
                g.fillRect(col * largEl,
                      lin * largEl + espLinha + ESPACO_TOPO,
                      largEl, largEl - espLinha * 2);
            }
        }
    }
```

Nem todos os valores contidos na grade serão desenhados. Os blocos que formam nosso labirinto são desenhados na cor azul, quando nosso personagem está no modo normal e, na cor cinza escuro, para indicar que nosso personagem entrou no modo super.

Na cor branca, temos nossas pastilhas e superpastilhas, que têm tamanhos diferentes e ocupam menos espaço que o bloco, e a linha por onde os fantasmas entram e saem da prisão.

Todos os desenhos levam em consideração o espaçamento superior, somando `ESPACO_TOPO`.

> **Dica**
>
> Consideramos o espaçamento apenas na pintura do desenho. Isso quer dizer que a posição original dos elementos não é alterada.
>
> Outra opção seria aplicar esse espaçamento nas posições dos elementos, mas neste caso ficaria mais complicada nossa interação com a `grade`, que é um array.

Essa é nossa grade com suas bordas na cor branca:

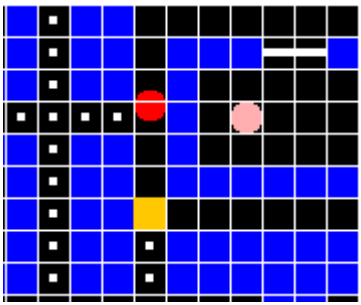

Fig. 6.5: Grade

Só para não esquecer, dentro do método `desenhar`, ainda temos que exibir os pontos do jogador e os personagens do jogo.

```
texto.desenha(g, "Pontos: " + pontos, 10, 20);

pizza.desenha(g);

for (Elemento el : inimigos) {
    if (el == null)
        continue;

    el.desenha(g);
}
```

Para trazer um bom desafio ao jogador, não basta encher nosso labirinto de inimigos. Precisamos que eles tenham inteligência suficiente para saírem e voltarem para a cela, que eles saibam quando ir atrás do nosso herói e quando fugir dele. Garanto-lhe que essa não será uma tarefa trivial, embora façamos tudo em um único método.

6.3 Um pouco de I.A. não faz mal

A Inteligência Artificial (*Artificial Intelligence* ou *AI*) nos jogos não precisa ser realmente inteligente, mas com certeza precisa passar a impressão de ser. Algumas variam de acordo com o nível do jogador, outras simplesmente tentam ser impiedosas e tem casos em que ela apenas segue um padrão.

Se o jogo tiver personagens não jogáveis (*non-player characters* ou *NPCs*), uma boa IA se torna vital. Quem nunca teve raiva de um NPC burro?

A IA é aplicada em diversas áreas da programação e, mesmo focando somente nas que estão presentes nos jogos, ainda assim precisaríamos de um livro ou mais, uma vez que sua aplicação inclui diversos tópicos, como: tomada de decisão, movimentação, estratégia, aprendizado e muito mais. Além de algoritmos específicos para determinados tipos de jogos, seja um jogo de corrida ou um de tabuleiro. Você pode ver mais sobre o assunto no livro *Artificial Intelligence for games*, de Ian Millington e John Funge.

A principal diferença entre os inimigos desse jogo em relação aos que já programamos nos capítulos anteriores é que se eles ficarem se movendo a esmo, eles podem, além de ficarem presos, não apresentar nenhuma ameaça para o jogador, deixando o jogo chato. Então, eles precisam ter perspicácia suficiente para perseguirem o jogador ou fugirem dele, sempre respeitando as regras do jogo, isso é, sem atravessar paredes ou coisa do tipo.

O que veremos a seguir é uma versão muito (mas muito) simplificada de *path finding*, algo do tipo "procurando o caminho". Se você ficou curioso e quer saber mais, entre em http://qiao.github.io/PathFinding.js/visual. Lá, você encontra uma versão bem legal onde pode testar diferentes algoritmos.

Para simplificar as coisas, o código ficou segmentando em cada um dos estados que nossos antagonistas podem ter, com exceção dos estados caçando e fugindo. Mesmo gerando certa repetição, essa separação nos per-

mite testar e entender melhor cada trecho. Tudo dentro do nosso método `atualizaDirecaoInimigos`, que começa assim:

```
if (foraDaTela(el))
    return;

int col = convertePosicaoIndice(el.getPx());
int lin = convertePosicaoIndice(el.getPy());

Direcao direcao = el.getDirecao();

// Variáveis auxiliares
Direcao tempDir = null;
int tempDx = 0, tempDy = 0;
int xCol = 0, yLin = 0;
```

No jogo original, um fantasma começa fora da jaula e outros três começam presos. Então, usamos somente dois neste jogo, um em cada modo.

```
if (el.getModo() == Pizza.Modo.PRESO) {
    if (el.getDirecao() == Direcao.SUL &&
                !validaDirecao(Direcao.SUL, el))
        el.setDirecao(Direcao.NORTE);

    else if (el.getDirecao() == Direcao.NORTE &&
                !validaDirecao(Direcao.NORTE, el))
        el.setDirecao(Direcao.SUL);

    else if (el.getDirecao() != Direcao.NORTE &&
                    el.getDirecao() != Direcao.SUL)
        el.setDirecao(Direcao.NORTE);

    if (temporizadorFantasma > 50)
        el.setModo(Pizza.Modo.ATIVO);
    else
        temporizadorFantasma++;
```

O inimigo está preso, mas não imóvel, ele anda na direção Norte até não poder mais, então troca para a direção oposta até não poder mais também.

Caso a direção não seja nem Norte, nem Sul, ela é definida como Norte, utilizando um temporizador para saber quando mudar o estado do inimigo para que ele saia da prisão.

```
} else if (el.getModo() == Pizza.Modo.ATIVO) {
    xCol = pontoFugaCol;
    yLin = pontoFugaLin;

    int colLarg = convertePosicaoIndice(el.getPx() +
                    el.getLargura() - el.getVel());
    int linAlt = convertePosicaoIndice(el.getPy() +
                    el.getAltura() - el.getVel());

    if (lin > yLin && validaDirecao(Direcao.NORTE, el))
        el.setDirecao(Direcao.NORTE);

    else if (lin < yLin && validaDirecao(Direcao.SUL, el))
        el.setDirecao(Direcao.SUL);

    else if (col < xCol && validaDirecao(Direcao.LESTE, el))
        el.setDirecao(Direcao.LESTE);

    else if (col > xCol && validaDirecao(Direcao.OESTE, el))
        el.setDirecao(Direcao.OESTE);

    else if (col == xCol && lin == yLin &&
            colLarg == xCol && linAlt == yLin) {
        el.setAtivo(true);
        el.setModo(Pizza.Modo.CACANDO);
    }
```

Primeiro, achamos a saída, cujas posições foram guardadas em `pontoFugaCol` e `pontoFugaLin`, depois verificamos se o personagem está na linha de cima ou de baixo em relação à linha do ponto de fuga, movendo-o na direção Norte ou Sul até ele chegar o mais próximo possível, então fazemos o mesmo para Leste e Oeste.

Quando finalmente ele se posiciona completamente na mesma linha e coluna do ponto de fuga (para isso, comparamos não somente a posição, mas

também largura e altura), ele fica ativo e pronto para caçar nosso jogador ou fugir dele.

```
} else if (el.getModo() == Pizza.Modo.CACANDO ||
            el.getModo() == Pizza.Modo.FUGINDO) {

    xCol = convertePosicaoIndice(pizza.getPx());
    yLin = convertePosicaoIndice(pizza.getPy());

    // Inverte posição para fugir
    if (el.getModo() == Pizza.Modo.FUGINDO) {
        xCol = xCol * -1;
        yLin = yLin * -1;
    }

    boolean perdido = rand.nextInt(100) == 35;

    if (el.isAtivo() && perdido) {
        tempDir = sorteiaDirecao();

    } else if (direcao == null) {
        direcao = sorteiaDirecao();

    } else if (direcao == Direcao.NORTE ||
               direcao == Direcao.SUL) {
        if (xCol < col && validaDirecao(Direcao.OESTE, el))
            tempDir = Direcao.OESTE;
        else if (xCol > col && validaDirecao(Direcao.LESTE, el))
            tempDir = Direcao.LESTE;

    } else {
        /* direcao = OESTE ou LESTE */
        if (yLin < lin && validaDirecao(Direcao.NORTE, el))
            tempDir = Direcao.NORTE;
        else if (yLin > lin && validaDirecao(Direcao.SUL, el))
            tempDir = Direcao.SUL;
    }

    if (tempDir != null && validaDirecao(tempDir, el))
```

```
            el.setDirecao(tempDir);
        else if (!validaDirecao(el.getDirecao(), el))
            el.setDirecao(sorteiaDirecao());
```

A única diferença entre o inimigo fugindo e o inimigo caçando é que, no primeiro modo, ele vai em direção ao jogador; já no segundo, ele obtém a direção do jogador e inverte-a, indo na direção oposta.

Para os inimigos não acabarem fazendo o mesmo movimento, uns dos outros, ou acabarem em uma perseguição impiedosa, de vez em quando fazemos o inimigo ir para qualquer lado, como se estivesse perdido.

A principal diferença aqui é que eles não ficam na mesma direção até colidirem com uma parede, eles estão sempre tentando mudar de direção no meio do caminho, mas não para a direção oposta; caso contrário, dependendo da posição do jogador, eles acabariam sem sair do lugar. Quando chegam a um beco sem saída, eles ficam lá até sortearem uma direção válida. Quando o jogador entra no modo super e colide com o inimigo, transforma-o em um fantasma.

```
} else if (el.getModo() == Pizza.Modo.FANTASMA) {
    xCol = pontoFugaCol;
    yLin = pontoFugaLin;

    if (direcao == Direcao.NORTE || direcao == Direcao.SUL) {
        if (xCol < col && validaDirecao(Direcao.OESTE, el))
            tempDir = Direcao.OESTE;
        else if (xCol > col && validaDirecao(Direcao.LESTE, el))
            tempDir = Direcao.LESTE;

    } else {
        if (yLin < lin && validaDirecao(Direcao.NORTE, el))
            tempDir = Direcao.NORTE;
        else if (yLin > lin && validaDirecao(Direcao.SUL, el))
            tempDir = Direcao.SUL;
    }

    if (tempDir != null && validaDirecao(tempDir, el))
        el.setDirecao(tempDir);
    else if (!validaDirecao(el.getDirecao(), el))
```

```
            el.setDirecao(trocaDirecao(el.getDirecao()));

        if (col == xCol && lin == yLin)
            el.setModo(Pizza.Modo.INATIVO);
```

No modo fantasma, o inimigo não oferece mais perigo para o jogador e tem de voltar para a prisão para se recuperar (mais precisamente, o ponto de volta). Primeiro, ele deve voltar ao ponto de fuga, já que dali só precisa ir em linha reta, cruzar a linha branca e chegar até o ponto de volta.

Para isso, ele usa parte do algoritmo usado na perseguição do jogador, mas sem ficar se perdendo no meio do caminho, sendo que se ele acabar em uma posição inválida na vertical, ele troca para uma direção na horizontal. Uma vez que cruzando a linha branca, ele volta ao seu aspecto normal e entra no modo inativo.

```
} else if (el.getModo() == Pizza.Modo.INATIVO) {
    xCol = pontoVoltaCol;
    yLin = pontoVoltaLin;

    if (lin > yLin && validaDirecao(Direcao.NORTE, el))
        el.setDirecao(Direcao.NORTE);

    else if (lin < yLin && validaDirecao(Direcao.SUL, el))
        el.setDirecao(Direcao.SUL);

    else if (col < xCol && validaDirecao(Direcao.LESTE, el))
        el.setDirecao(Direcao.LESTE);

    else if (col > xCol && validaDirecao(Direcao.OESTE, el))
        el.setDirecao(Direcao.OESTE);

    else if (col == xCol && lin == yLin)
        el.setModo(Pizza.Modo.PRESO);
}
```

Essa é a mesma lógica do modo ativo, e o código quase idêntico. Mas em vez de chegar na linha e coluna do ponto de fuga, estamos mirando na linha e coluna do ponto de volta e, uma vez lá, o inimigo volta ao modo preso, iniciando novamente o ciclo sem fim.

Antes de o método `atualizaDirecaoInimigos` acabar, precisamos atualizar a direção do personagem caso ela seja válida.

```
if (validaDirecao(el.getDirecao(), el)) {
    if (el.getDirecao() == Direcao.NORTE)
        tempDy = -1;
    else if (el.getDirecao() == Direcao.SUL)
        tempDy = 1;
    else if (el.getDirecao() == Direcao.OESTE)
        tempDx = -1;
    else if (el.getDirecao() == Direcao.LESTE)
        tempDx = 1;
}

el.setDx(tempDx);
el.setDy(tempDy);
```

Vale notar que o algoritmo para o inimigo se mover leva em consideração que, no labirinto, ele só pode andar no eixo X ou Y, mas dentro da jaula ele tem mais espaço. Devemos levar isso em conta para ele não acabar se movendo na diagonal. Não esqueça de considerar isso ao criar seus próprios labirintos.

Os dois últimos métodos auxiliares que precisamos ver realmente não fazem nada de mais.

```
private Direcao trocaDirecao(Direcao direcao) {
    if (direcao == Direcao.NORTE)
        return Direcao.OESTE;
    else if (direcao == Direcao.OESTE)
        return Direcao.SUL;
    else if (direcao == Direcao.SUL)
        return Direcao.LESTE;
    else
        return Direcao.NORTE;
}

private Direcao sorteiaDirecao() {
    return Direcao.values()[rand.
                            nextInt(Direcao.values().length)];
}
```

O primeiro método troca a direção sem invertê-la, apenas passando da horizontal para a vertical, ou da vertical para horizontal. O segundo apenas pega uma direção de forma aleatória. Assim, abordamos os principais pontos do código desse jogo e talvez, depois disso, você pegue mais leve com o desenvolvedor quando der de cara com um bug de IA.

6.4 Codificando o jogo

Temos bastante código aqui para que nosso herói possa percorrer seu labirinto atrás de comida, e mais código para que ele seja perseguido por seus inimigos. Vale lembrar que uma versão pronta com três níveis de dificuldades diferentes lhe espera em: https://github.com/logicadojogo/fontes/tree/master/Capo6.

Na classe `Jogo.java`, apenas o tamanho da tela muda (448 de largura por 550 de altura) em relação ao jogo anterior. Seguindo o padrão, toda a lógica do jogo ficou em `JogoCenario.java`.

Novas classes `Nivel.java` e `Pizza.java` foram criadas, e temos poucas mudanças em `InicioCenario.java`, que permite ao jogador escolher o modo de jogo (fácil, normal e difícil).

Não fizemos aqui

- Ter a família de quatro fantasmas completa;
- Reiniciar o cenário ao comer todas as pastilhas;
- Criar uma cópia do cenário para não perder a configuração inicial.

Melhore você mesmo

- Adicionar frutas que aumentem a pontuação do jogador.

6.5 Resumo

Começamos pelo labirinto, embora ele tenha sido criado com base no código já discutido no capítulo 4. Desta vez, utilizamos mais opções com valores que

vão de 0 até 9, guardados em constantes.

Carregamos o jogador e os inimigos em duas etapas: primeiro configuramos aspectos básicos, e depois percorremos o nível em busca das posições iniciais, ponto de fuga e de volta, como também somamos todas as pastilhas que o jogador terá de comer para concluir o jogo.

Trabalhamos com elementos em eixo colidindo com elementos no array, convertendo índice em posição, e posição em índice. Para facilitar a movimentação, usamos pontos cardeais na direção dos personagens. Ao validar suas movimentações, temos de considerar a posição e o tamanho deles e verificar a colisão na grade, semelhante à detecção de colisão entre dois elementos.

Vimos que nossos fantasmas têm seis estágios durante o jogo: preso, ativo, caçando, fugindo, fantasma e inativo. E nosso herói, mesmo que temporariamente, entra no modo superpizza. Temos praticamente um algoritmo para cada estado do inimigo, seja para eles saírem e voltarem para a cela, ou perseguirem e fugirem do nosso herói. Mesmo assim, apenas arranhamos a superfície do assunto sobre Inteligência Artificial.

Mesmo depois de tanto código, o jogo não ficou tão parecido com o original como esperávamos, por isso, faremos melhorias e usaremos imagens para deixar o jogo mais bonito. Não perca o próximo capítulo!

Capítulo 7

Come-come e os bitmaps

Até agora, usamos código para criar nossos desenhos, mas trabalhar com imagens mais complexas e detalhadas exigiria muito mais programação. A forma mais fácil de utilizarmos desenhos complexos dentro dos nossos jogos é criá-los usando uma ferramenta apropriada, por exemplo, Gimp, Photoshop ou Fireworks, e importá-los de um arquivo externo para desenhá-los na tela.

Neste capítulo, trabalharemos com:

- Imagens;
- Sprites;
- Tiles.

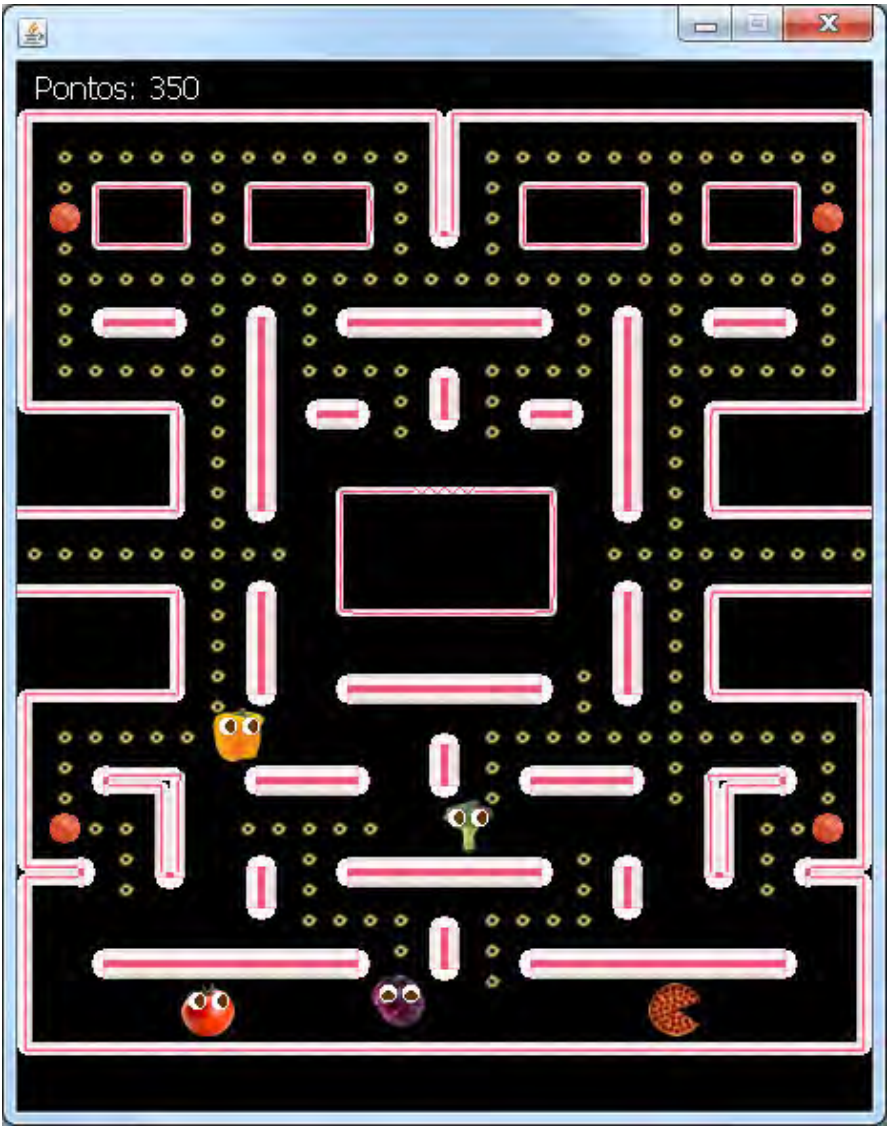

Fig. 7.1: Come-come com imagens

O Java facilita nossa vida com o objeto `javax.swing.ImageIcon`, que aceita diversos formatos como BMP (*Bitmap*), JPG e PNG, mas utilizaremos somente o PNG (*Portable Network Graphics*), que suporta fundo transparen-

tes (canal alfa).

Aqui, faremos um jogo temático, e o tema escolhido é o mundo da pizza, onde nosso protagonista será uma pizza propriamente dita e seus inimigos são legumes. As bordas se tranformaram em canudos, as pastilhas pequenas viraram azeitonas e as maiores, pepperoni.

Como mantivemos as mesmas dimensões do jogo anterior, as imagens estão pequenas, e pode ser necessário apelar para a imaginação para conseguir distinguir as coisas.

7.1 Uma imagem vale mais do que mil linhas de código

> **Sprites e Tiles**
>
> **Sprite** é uma imagem que agrupa diversas imagens, geralmente com a mesma altura e largura, e são usados para representar o estado do personagem ou criar animações.
>
> **Tile** é parecido com o sprite, mas são usados para montar cenários ou objetos estáticos do jogo. As partes de um tile podem formar cenários diferentes dependendo da combinação, ou usando mais ou menos partes para formar objetos com diferentes tamanhos.
>
> Recomendo que procure imagens no Google usando as seguintes palavras: "sprites sonic" e "tiles super mario bros".

Desenhar imagens no nosso jogo é realmente muito fácil. Nós precisamos de uma imagem, um objeto `ImageIcon` e chamar o método `drawImage` do objeto `Graphics2D`.

```
ImageIcon azeitona = new ImageIcon("imagens/azeitona.png");

public void desenhar(Graphics2D g) {
    g.drawImage(azeitona.getImage(), 10, 30, null);
}
```

Para carregar um arquivo externo, precisamos do seu caminho. Neste caso, passamos o caminho relativo, já que a pasta *imagens* está no mesmo diretório do nosso projeto, semelhante à forma como carregamos arquivos de áudio no capítulo 5.

Para desenhá-la na tela, usamos o método `drawImage`, mas poderíamos usar o método `paintIcon` do próprio `ImageIcon`: `azeitona.paintIcon(null, g, 10, 30)`. O primeiro parâmetro do método `drawImage` é um objeto da classe abstrata `Image`, que o `ImageIcon` nos fornece.

Depois, temos as posições em eixo onde a imagem será desenhada. O último parâmetro é um objeto `ImageObserver`, que não usaremos neste livro, então passamos `null`.

O método `paintIcon` pede como primeiro argumento um objeto `Component` para usar como `ImageObserver`, depois o objeto `Graphics` seguido pelas posições nos eixos. Nos dois exemplos, desenhamos a imagem na posição 10 do eixo X, e 30 do eixo Y.

Dica

Se quiser trabalhar diretamente com o objeto `Image`, pode usar o `java.awt.Toolkit` para carregá-las:

```
public void desenhar(Graphics2D g) {
    Toolkit tk = Toolkit.getDefaultToolkit();
    Image imagem = tk.getImage("imagens/azeitona.png");
    g.drawImage(imagem, 0, 0, null);

}
```

Nossas imagens (`azeitona.png` e `pepperoni.png`) têm o mesmo tamanho, 18x18 pixels, mas a área que a azeitona ocupa do tamanho total da imagem é de apenas de 8x8. Assim, fica mais fácil para desenhar, já que não teremos de considerar as diferenças de tamanho.

Vale acrescentar que 18x18 é o tamanho do corredor que os personagens percorrem no labirinto.

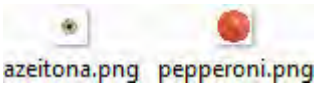

Fig. 7.2: Imagens simples

Dessa forma, desenhamos as azeitonas e os pepperonis. Quando falamos de uma única imagem em um arquivo, é simples assim, mas quando um arquivo representa mais de uma imagem (um sprite), o código fica mais interessante. Veja um exemplo na figura:

Fig. 7.3: Sprite simples

Esses são os quatro estágios do nosso personagem em um único arquivo. Se por acaso utilizarmos o método citado, o jogador verá esse trem de pizzas mas, em vez disso, queremos que ele veja um formato de pizza de cada vez, dando a impressão de animação. Para fazermos isso da forma correta e sem precisar usar várias imagens, teremos de usar um método da classe `Graphics` um pouco maior:

```
public abstract boolean drawImage(Image img,
int dx1, int dy1, int dx2, int dy2,
int sx1, int sy1, int sx2, int sy2,
                                  ImageObserver observer);
```

Essa declaração de método veio direto da documentação, sendo que os quatro primeiros inteiros (`int`) são referentes ao destino da imagem e os quatro últimos referentes à imagem de origem. Em vez de reproduzir a explicação vinda da própria documentação, vamos explicar com exemplos.

Para começar, imagine os quatro primeiros parâmetros como uma moldura onde queremos que nossa imagem apareça na tela.

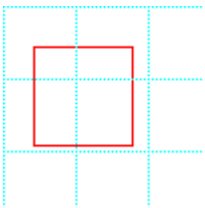

Fig. 7.4: Moldura

Neste caso, nossa tela tem 100 pixels de largura e altura, e nossa moldura 50 x 50px. Usamos os primeiros parâmetros para posicionar e dimensionar a moldura na tela, representada pelo quadrado vermelho. Os dois primeiros são a origem, e os dois últimos o destino do quadrado, e não a largura e altura. Isso quer dizer que devemos levar em conta as posições iniciais e somá-las às posições finais.

Os quatro últimos são referentes aos pontos de origem e tamanho na imagem:

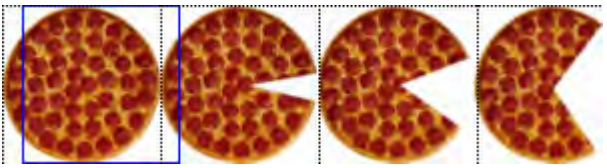

Fig. 7.5: Marcação de recorte

As dimensões da nossa imagem (`sprite_simples.png`) são: 300 x 79px, e nosso quadrado azul tem 79 x 79px com deslocamento de 10 pixels no eixo X. Dessa forma, aplicamos um recorte na imagem, já que somente o que estiver dentro do quadrado será desenhado.

Nosso código para recortar a imagem e desenhá-la na tela nos eixos 15x e 20y ficaria assim:

```
ImageIcon img = new ImageIcon("imagens/sprite_simples.png");

@Override
```

```
public void desenhar(Graphics2D g) {
    g.drawImage(img.getImage(),
            15, 20, 15 + 79, 20 + 79,
            10, 0, 10 + 79, 0 + 79, null);

}
```

Observe que somamos os valores usados no deslocamento para definir as posições finais da moldura e da imagem, tendo como resultado a moldura na posição 15x, 20y, e a imagem 10x, 0y – ambas com largura e altura de 79px.

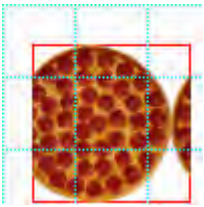

Fig. 7.6: Resultado do recorte

Embora possa parecer confuso tantos parâmetros, usaremos tantas vezes esse método durante o capítulo que você acabará se acostumado, assim espero. Lembre-se de que temos dois parâmetros para representar as posições iniciais da imagem, e dois para representar as posições de destino, e não largura e altura.

Se esquecer de levar em consideração as posições inicias na hora de definir as posições de destino, a chamada do método escalonará a imagem (se ela ficar distorcida, você já sabe o motivo).

Um exemplo prático disso é quando fazemos uma seleção com o mouse: você clica, segura definindo as posições iniciais e move o mouse em qualquer direção para definir a posição de destino. A vantagem disso no método `drawImage` é que podemos inverter a imagem simplesmente invertendo os parâmetros:

```
ImageIcon img = new ImageIcon("imagens/copo_mshake.png");
// Normal
g.drawImage(img.getImage(), 0, 0, 78, 138, 0, 0, 78, 138, null);
```

```
// Invertido
g.drawImage(img.getImage(), 0, 0, 78, 138, 78, 0, 0, 138, null);
```

Na primeira chamada, o ponto de origem é 0, o ponto final do eixo X é 78 e o eixo Y é 138, tanto na posição da moldura quanto na da imagem.

Na segunda chamada do método, invertemos as posições iniciais do eixo X, ficando com 78 na origem e 0 na final, espelhando a imagem.

Fig. 7.7: Imagem espelho

Já consegue imaginar o que aconteceria se invertêssemos o eixo Y? Isso mesmo, a imagem ficaria de ponta cabeça.

Mas não se preocupe, esses e outros feitos dos métodos de desenho do Java não serão usados neste jogo, apenas sentimos que esse era um ponto a ser visto.

Por outro lado, o sprite que usaremos no jogo é um pouco maior (e mais complicado). Ele contém as quatro pizzas em todas as direções em que o jogador pode ir durante o jogo:

Fig. 7.8: Pizzas pra que te quero

Trabalharemos com sprites como se eles fossem uma grade com linhas e colunas. Neste caso, o sprite do nosso herói tem 4 linhas e 4 colunas.

A primeira e segunda linha correspondem ao movimento do jogador para a direita (Leste) e para esquerda (Oeste). A terceira e quarta linha, o movimento para cima (Norte) e para baixo (Sul).

Enquanto o jogador estiver em movimento, percorreremos as colunas do sprite, na linha referente à direção escolhida, animando a movimentação do personagem. Vamos ao exemplo:

```
@Override
public void desenhar(Graphics2D g) {
    ImageIcon img = new ImageIcon("imagens/sprite_pizza.png");

    int coluna = 3;
    int linha = 0;

    int largMoldura = img.getIconWidth() / 4;
```

```
    int altMoldura = img.getIconHeight() / 4;

    int largImg = largMoldura * coluna;
    int altImg = altMoldura * linha;

    g.drawImage(img.getImage(),
            0, 0, largMoldura, altMoldura,
            largImg, altImg, largImg + largMoldura,
            altImg + altMoldura, null);

}
```

Utilizamos nossa imagem `sprite_pizza.png`, onde temos 4 linhas e 4 colunas. As colunas e linhas são referenciadas como o índice de um array, começando no 0. Ao definir `coluna` com o valor 3 e `linha` com 0, carregaremos a figura da última coluna da primeira linha.

Mesmo sabendo que todas as figuras dentro do sprite têm o mesmo tamanho (28px), não utilizaremos valores fixos, assim fica mais fácil fazer alterações na imagem.

Para acharmos a largura e altura da moldura, basta dividirmos a largura e altura da imagem, obtidas com `getIconWidth()` e `getIconHeight()` respectivamente, pela quantidade total de linhas e colunas. E para acharmos a posição correta da coluna e linha que queremos desenhar, multiplicamos `largMoldura` pela `coluna`, e `altMoldura` pela `linha`.

Em termos mais práticos, nosso sprite tem 112 x 112 pixels, então nós temos uma moldura de 28 x 28 pixels (112 / 4), que é o tamanho de cada figura. Nossa coluna começa no pixel 84 (28 * 3), nossa linha no pixel 0 (28 * 0), e terminam em 112 (84 + 28) e 28 (0 + 28), respectivamente.

Mesmo com tanta matemática, ainda não estamos considerando o posicionamento na tela, mas desta forma fica mais fácil pegar parte da imagem, basta alterar o valor de `linha` e `coluna`.

Aplicaremos um código semelhante em outras imagens do jogo:

Fig. 7.9: Olhos fantasma – 1 coluna e 4 linhas

Além de indicar a direção em que o fantasma está indo, os olhos são a única coisa que sobra quando ele é devorado pela nossa superpizza.

Fig. 7.10: Inimigos – 4 colunas e 2 linhas

Tomate, Repolho-roxo, Pimentão amarelo e Brócolis representam cada um dos nossos personagens. Apenas invertemos as cores para indicar em que estado estão (caçando ou fugindo), e ainda não temos 100% de certeza se Tomate é um legume.

Fig. 7.11: Cenário – 5 linhas e 9 colunas

Nosso cenário é composto por diversas imagens. Cada parte dele vem de um pedaço desta figura em vez de diversas imagens separadas conforme veremos adiante, bem como mais alguns truques conforme melhoramos e evoluímos nosso jogo.

7.2 Jogando o código

Faz alguns jogos que não mexemos no nosso código base, então, vamos tirar a poeira da nossa classe `Elemento`, adicionando três novas propriedades:

```
public class Elemento {

    private int dx;
    private int dy;
    private ImageIcon imagem;

    public void desenha(Graphics2D g) {
        if (imagem == null) {
            g.setColor(cor);
            g.fillRect(px, py, largura, altura);
        } else {
            g.drawImage(imagem.getImage(), px, py, null);
        }
    }
...
```

Tanto `dx` quanto `dy` já são nossos conhecidos, utilizamos em quase todos os jogos para direcionar nossos personagens na tela. A maioria dos nossos elementos terá uma imagem, então nada melhor do que usarmos uma variável para guardá-la em nossa classe `Elemento`, deixando nosso método `desenha` com uma implementação padrão que exiba algum desenho caso nosso objeto `imagem` seja nulo, ou desenhe a própria imagem.

Ocultamos aqui as demais variáveis e métodos, inclusive os `gets` e `sets` das propriedades novas.

> **Dica**
>
> Nossas variáveis `dx` e `dy` são do tipo `int`, mas poderiam ser do tipo `byte`, já que só possuem três valores: -1, 0 e 1.

Na primeira versão do jogo, usamos o mesmo objeto para todos os personagens, mas desta vez eles serão atualizados e desenhados de forma dife-

rentes, além de terem propriedades distintas. Nosso herói continuará sendo representado pela classe `Pizza`, nossos inimigos pela classe `Legume`.

```
public class Pizza extends Elemento {

    private Direcao direcao = Direcao.OESTE;

    public Pizza() {
        super(0, 0, 16, 16);
        setImagem(new ImageIcon("imagens/sprite_pizza.png"));
    }

    private int linha;
    private int coluna;
```

O controle do personagem não altera `linha` e `coluna` diretamente, isso é feito no método `atualiza`, que posiciona na linha correta de acordo com a direção definida.

```
@Override
    public void atualiza() {
        incPx(getVel() * getDx());
        incPy(getVel() * getDy());

        if (getDx() == 1)
            linha = 0;
        else if (getDx() == -1)
            linha = 1;
        else if (getDy() == -1)
            linha = 2;
        else if (getDy() == 1)
            linha = 3;

        if (getDx() + getDy() != 0)
            coluna++;

        if (coluna > 3)
            coluna = 0;
    }
```

```
...
}
```

Lembrando de que a direção real é controlada por `dx` e `dy`, e se o personagem estiver em movimento, um ou outro terá o valor 0. Quando os dois estiverem com o valor 0, o personagem estará parado, então não mudamos a coluna do sprite.

O método de desenho é semelhante ao já visto anteriormente, o que muda agora é que consideramos as propriedades do jogo, como a posição do personagem na tela e o espaçamento do cenário, por exemplo.

```
@Override
public void desenha(Graphics2D g) {

    int pX = getPx() - 6;
    int pY = getPy() + JogoCenario.ESPACO_TOPO - 6;

    // Largura e altura da moldura
    int largMoldura = getImagem().getIconWidth() / 4;
    int altMoldura = getImagem().getIconHeight() / 4;

    // Largura e altura do recorte da imagem
    int largImg = largMoldura * coluna;
    int altImg = altMoldura * linha;

    g.drawImage(getImagem().getImage(),
            pX, pY, pX + largMoldura, pY + altMoldura,
            largImg, altImg,
            largImg + largMoldura,
            altImg + altMoldura, null);

}
```

Reduzimos a margem esquerda e, levando em consideração o espaço entre a grade e o topo da tela, reduzimos a margem superior (subtraindo 6 pixels em cada eixo), isso para nosso personagem ter 12 pixels de desenho a mais que em sua versão anterior.

O tamanho do elemento (e de cada espaço na grade) continua sendo 16 por 16px, mas cada figura do nosso sprite tem 28 por 28 pixels. Este é o truque

para usarmos uma imagem maior: aumentamos o tamanho do personagem e diminuímos o tamanho das paredes do labirinto, como veremos mais à frente. As variáveis `pX` e `pY` são referentes à posição do elemento na tela, onde posicionaremos nossa moldura.

Os inimigos leguminosos dão um pouco mais de trabalho:

```
public class Legume extends Elemento {

    public enum Modo {
        PRESO, ATIVO, INATIVO, FANTASMA, CACANDO, FUGINDO;
    }

    public enum Tipo {
        VERMELHO, ROXO, AMARELO, VERDE;
    }

    private Tipo tipo;
    private Modo modo = Modo.PRESO;
    private Direcao direcao = Direcao.OESTE;

    private int linha;
    private int coluna;
    private int lnOlhos; // Linha olhos
```

Os modos que nosso objeto pode assumir continuam sendo os mesmos do jogo anterior, mas agora ele tem um `Tipo` que usamos para saber qual coluna do sprite queremos, já que cada uma representa um personagem em seu modo caçando e fugindo.

```
private static ImageIcon olhos;
private static ImageIcon sprite;

static {
    olhos = new ImageIcon("imagens/olhos.png");
    sprite = new ImageIcon("imagens/sprite_inimigos.png");
}

public Legume(Tipo tipo) {
    super(0, 0, 16, 16);
```

```
        this.tipo = tipo;
        this.coluna = tipo.ordinal();
    }
```

As imagens para os olhos e para os inimigos são estáticas, pertencem à classe, e não à instância dela. O melhor seria ter uma classe separada para gerenciar as imagens do jogo, mas assim mantemos as coisas simples e reaproveitamos as mesmas instâncias do `ImageIcon` para os quatro inimigos.

Nossa imagem `olhos` tem apenas uma coluna, então só precisamos nos preocupar com as linhas que variam dependendo da direção do personagem, semelhante ao que fazemos com nosso herói.

```
@Override
    public void atualiza() {
        incPx(getVel() * getDx());
        incPy(getVel() * getDy());

        if (getDx() == -1)
            lnOlhos = 0;
        else if (getDx() == 1)
            lnOlhos = 1;
        else if (getDy() == -1)
            lnOlhos = 2;
        else if (getDy() == 1)
            lnOlhos = 3;

        if (modo == Modo.FUGINDO)
            linha = 1;
        else
            linha = 0;
    }
}
```

Utilizamos uma imagem diferente se o inimigo estiver fugindo, representada pela segunda linha do sprite. Não existe mistério no método `desenha`.

Vale notar que, quando o inimigo vira um fantasma, não desenhamos a imagem principal, logo, veremos apenas um par de olhos vagando pela tela.

```
@Override
public void desenha(Graphics2D g) {
    int pX = getPx() - 6;
    int pY = getPy() + JogoCenario.ESPACO_TOPO - 6;

    int largMoldura = sprite.getIconWidth() / 4;
    int altMoldura = sprite.getIconHeight() / 2;

    int largImg = largMoldura * coluna;
    int altImg = altMoldura * linha;

    if (modo != Modo.FANTASMA)
        g.drawImage(sprite.getImage(),
                pX, pY,
                pX + largMoldura, pY + altMoldura,
                largImg, altImg, largImg + largMoldura,
                altImg + altMoldura, null);

    largMoldura = olhos.getIconWidth();
    altMoldura = olhos.getIconHeight() / 4;
    altImg = altMoldura * lnOlhos;

    g.drawImage(olhos.getImage(),
                pX, pY,
                pX + largMoldura, pY + altMoldura,
                0, altImg, largMoldura,
                altImg + altMoldura, null);
}
```

Embora tenhamos trabalho dobrado no método `desenha`, criar um sprite com todas as combinações de figuras possíveis ficaria inviável e, como podemos ver, fazer mesclas de imagens é simplesmente desenhar uma sobre a outra. Lembrando de que a primeira a ser desenhada ficará por baixo.

Nossa imagem `olhos.png` tem largura e altura própria para que cada figura fique na posição correta no "rosto" do personagem.

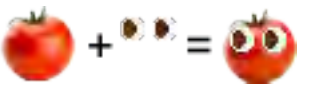

Fig. 7.12: Sobrepondo imagens

Depois de usarmos um sprite para termos um par de olhos olhando em cada direção, chegou a hora de você perguntar se usar imagens previamente *rotacionadas* é o único jeito de fazermos isso. Para nossa sorte, não, essa não é a única forma.

Mas como isso não é um assunto trivial, veremos no próximo jogo. E agora com 5 linhas e 9 colunas, os tiles que formam o cenário do nosso jogo:

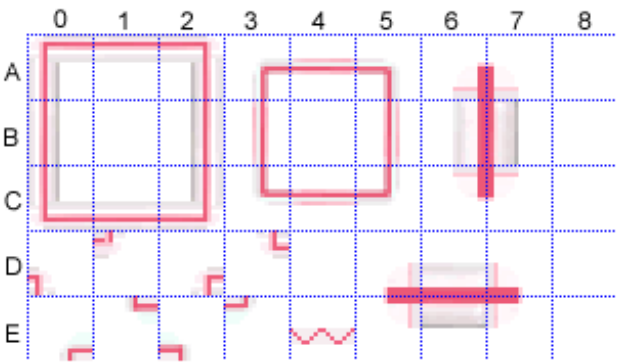

Fig. 7.13: Cenário em partes

Temos um novo problema aqui. Na primeira versão do jogo, nossa grade só precisava de um único valor para representar as paredes do labirinto, porém, nesta versão, precisaremos armazenar um valor que indique a coluna e linha referente à figura que queremos em nossa imagem `tiles_cenario.png`.

Adotaremos uma solução bem simples utilizando a matemática. Primeiramente, todos os códigos que não formem as paredes do labririnto passarão a ter um valor negativo. Depois, usaremos valores cuja divisão e resto da divisão por 10 nos indique a coluna e a linha. Tome, por exemplo, o valor 46, que

dividido por 10 resulta em 4,6. Assim, sabemos que 4 é a linha que queremos, e 6 a coluna.

```
...
int valor = grade[lin][col];

if (valor >= 0) {

    int linha = 0;
    int coluna = valor;

    if (valor > 9) {
        linha = valor / 10;
        coluna = valor % 10;
    }
...
} else {
// Outros itens do cenário
...
}
```

Dessa forma, nós sabemos que, se o valor da grade for maior que 9, então a linha será o valor dividido por 10, e a coluna será o resto da divisão por 10.

Olhando novamente a figura 7.13, note que cada espaço na imagem (quadrados pontilhados) tem 16 pixels de largura e altura, mas os desenhos não ocupam o espaço todo. Por isso, nossos personagens, mesmo sendo maiores que o tamanho padrão, circulam o labirinto sem sobrepor o cenário.

Para facilitar a criação de níveis, as paredes da primeira linha começam com a letra `A` e vão de 0 até 7, da segunda linha com a letra `B` e vão de 10 até 17, e assim por diante. Logo, D6 tem o valor de 36, que equivale à linha 3 e coluna 6 da imagem. Vale lembrar que nossa contagem começa em 0. Utilizamos de 0 a 7 já que não temos a coluna 9, e deixamos a coluna 8 caso você queira fazer modificações.

```
public static final int A0 = 0;
public static final int A1 = 1;
...
public static final int E0 = 40;
```

```
public static final int E2 = 42;
public static final int LN = 44; // E4
public static final int E5 = 45;
public static final int E6 = 46;
public static final int E7 = 47;
```

A constante `LN` poderia ser `E4`, mas foge a regra para facilitar a leitura em outras partes do código, assim como nossas antigas variáveis com valores novos:

```
public static final int BL = -1;  /** Bloco */
public static final int CN = -2;  /** Comida normal */
public static final int EV = -3;  /** Espaco vazio */
public static final int PI = -4;  /** Ponto inicial do jogador */
public static final int SC = -5;  /** Super comida */
public static final int P1 = -6;  /** Ponto inicial inimigo 1 */
public static final int P2 = -7;  /** Ponto inicial inimigo 2 */
public static final int P3 = -8;  /** Ponto inicial inimigo 1 */
public static final int P4 = -9;  /** Ponto inicial inimigo 2 */
public static final int PF = -10; /** Ponto de Fuga */
public static final int PV = -11; /** Ponto de Volta */
```

E tudo junto, formando um belo cenário, ficou assim:

```
{ A0, A1, A1, A1, A1, A1, A1, A1, A1, A1, A1, A1, A1, A2, A0, A1, A1, A1, A1, A1, A1, A1, A1, A1, A1, A1, A1, A2 },
{ B0, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, B2, B0, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, B2 },
{ B0, CN, A3, A4, A4, A5, CN, A3, A4, A4, A4, A5, CN, B2, B0, CN, A3, A4, A4, A4, A5, CN, A3, A4, A4, A5, CN, B2 },
{ B0, SC, B3, EV, EV, B5, CN, B3, EV, EV, EV, B3, CN, B2, B0, CN, B3, EV, EV, EV, B5, CN, B3, EV, EV, B5, SC, B2 },
{ B0, CN, C3, C4, C4, C5, CN, C3, C4, C4, C4, C5, CN, C6, C7, CN, C3, C4, C4, C4, C5, CN, C3, C4, C4, C5, CN, B2 },
{ B0, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, B2 },
{ B0, CN, D5, D6, D6, D7, CN, A6, A7, CN, D5, D6, D6, D6, D6, D6, D7, CN, A6, A7, CN, D5, D6, D6, D7, CN, B2 },
{ B0, CN, E5, E6, E6, E7, CN, B6, B7, CN, E5, E6, E6, E6, E6, E6, E7, CN, B6, B7, CN, E5, E6, E6, E7, CN, B2 },
{ B0, CN, CN, CN, CN, CN, CN, B6, B7, CN, CN, CN, CN, A6, A7, CN, CN, CN, CN, CN, B6, B7, CN, CN, CN, CN, CN, B2 },
{ C0, C1, C1, C1, C1, D0, CN, B6, B7, D5, D6, D7, CN, B6, B7, CN, D5, D6, D7, B6, B7, CN, D2, C1, C1, C1, C1, C2 },
{ EV, EV, EV, EV, EV, B0, CN, B6, B7, E5, E6, E7, CN, C6, C7, CN, E5, E6, E7, B6, B7, CN, B2, EV, EV, EV, EV, EV },
{ EV, EV, EV, EV, EV, B0, CN, B6, B7, EV, EV, EV, EV, PF, P1, EV, EV, EV, EV, B6, B7, CN, B2, EV, EV, EV, EV, EV },
{ EV, EV, EV, EV, EV, B0, CN, B6, B7, EV, A3, A4, A4, LN, LN, A4, A4, A5, EV, B6, B7, CN, B2, EV, EV, EV, EV, EV },
{ A1, A1, A1, A1, A1, D1, CN, C6, C7, EV, B3, EV, EV, EV, EV, EV, EV, B3, EV, C6, C7, CN, D3, A1, A1, A1, A1, A1 },
{ CN, CN, CN, CN, CN, CN, CN, CN, CN, EV, B3, EV, EV, PV, EV, EV, EV, B3, EV, CN, CN, CN, CN, CN, CN, CN, CN, CN },
{ C1, C1, C1, C1, C1, D0, CN, A6, A7, EV, B3, P2, EV, P3, EV, P4, EV, B3, EV, A6, A7, CN, D2, C1, C1, C1, C1, C1 },
{ EV, EV, EV, EV, EV, B0, CN, B6, B7, EV, C3, A4, A4, A4, A4, A4, A4, C5, EV, B6, B7, CN, B2, EV, EV, EV, EV, EV },
{ EV, EV, EV, EV, EV, B0, CN, B6, B7, EV, EV, EV, EV, PI, EV, EV, EV, EV, EV, B6, B7, CN, B2, EV, EV, EV, EV, EV },
{ EV, EV, EV, EV, EV, B0, CN, B6, B7, CN, D5, D6, D6, D6, D6, D6, D6, D7, CN, B6, B7, CN, B2, EV, EV, EV, EV, EV },
{ A0, A1, A1, A1, A1, D1, CN, C6, C7, CN, E5, E6, E6, E6, E6, E6, E6, E7, CN, C6, C7, CN, D3, A1, A1, A1, A1, A2 },
{ B0, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, A6, A7, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, B2 },
{ B0, CN, D5, C1, C1, E2, CN, D5, D6, D6, D6, D7, CN, B6, B7, CN, D5, D6, D6, D6, D7, CN, E0, C1, C1, D7, CN, B2 },
{ B0, CN, E5, A1, A2, B0, CN, E5, E6, E6, E6, E7, CN, C6, C7, CN, E5, E6, E6, E6, E7, CN, B2, A0, A1, E7, CN, B2 },
{ B0, SC, CN, CN, B2, B0, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, B2, B0, CN, CN, SC, B2 },
{ C0, C1, D7, CN, B2, B0, CN, A6, A7, CN, D5, D6, D6, D6, D6, D6, D6, D7, CN, A6, A7, CN, B2, B0, CN, D5, C1, C2 },
{ A0, A1, E7, CN, C6, C7, CN, B6, B7, CN, E5, E6, E6, E6, E6, E6, E7, CN, B6, B7, CN, C6, C7, CN, E5, A1, A2 },
{ B0, CN, CN, CN, CN, CN, CN, C6, C7, CN, CN, CN, CN, A6, A7, CN, CN, CN, CN, C6, C7, CN, CN, CN, CN, CN, CN, B2 },
{ B0, CN, D5, D6, D6, D6, D6, D6, D6, D6, D7, CN, B6, B7, CN, D5, D6, D6, D6, D6, D6, D6, D6, D7, CN, B2 },
{ B0, CN, E5, E6, E6, E6, E6, E6, E6, E6, E7, CN, C6, C7, CN, E5, E6, E6, E6, E6, E6, E6, E6, E7, CN, B2 },
{ B0, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, CN, B2 },
{ C0, C1, C1, C1, C1, C1, C1, C1, C1, C1, C1, C1, C1, C1, C1, C1, C1, C1, C1, C1, C1, C1, C1, C1, C1, C2 } };
```

Fig. 7.14: Novo cenário

O código da classe `JogoCenario.java` sofreu algumas mudanças, mas a maioria se refere à separação dos personagens na classe `Pizza` e `Legume`. Então, vamos nos focar nas mudanças mais importantes.

Começamos pelo método `desenhar`, que faz todo o trabalho que temos manipulando imagem valer a pena.

```
for (int lin = 0; lin < grade.length; lin++) {
    for (int col = 0; col < grade[0].length; col++) {
        int valor = grade[lin][col];

        if (valor >= 0) {

            int linha = 0;
            int coluna = valor;

            if (valor > 9) {
                linha = valor / 10;
                coluna = valor % 10;
            }
```

```
            int pX = col * largEl;
            int pY = lin * largEl + ESPACO_TOPO;
```

Valores maiores ou iguais a zero são as partes do cenário que compõem o labirinto, e se o valor for maior que 9, sabemos que, para achar a linha e coluna, temos de dividir por 10.

```
int largMoldura = cenario.getIconWidth() / 9;
int altMoldura = cenario.getIconHeight() / 5;

int largImg = largMoldura * coluna;
int altImg = altMoldura * linha;

g.drawImage(cenario.getImage(), pX, pY,
            pX + largMoldura, pY + altMoldura,
            largImg, altImg,
            largImg + largMoldura, altImg + altMoldura, null);
```

Nosso `cenario` (`tiles_cenario.png`) tem 9 colunas e 5 linhas, fora isso, é desenhado como qualquer outro sprite do jogo. Se o valor for menos que 0, verificamos se é uma comida normal ou supercomida.

```
} else {
    if (valor == Nivel.CN)
        g.drawImage(azeitona.getImage(),
                col * largEl, lin * largEl + ESPACO_TOPO, null);
    else if (valor == Nivel.SC)
        g.drawImage(pepperoni.getImage(),
                col * largEl, lin * largEl + ESPACO_TOPO, null);
}

}
}

texto.desenha(g, "Pontos: " + pontos, 10, 20);
pizza.desenha(g);

for (Elemento el : inimigos) {
    if (el == null)
```

```
            continue;

        el.desenha(g);
    }
```

Outro método modificado que merece um pouco mais de atenção é o `validaMovimento`.

```
    private boolean validaMovimento(Elemento el, int dx, int dy) {
        // Próxima posição x e y
        int prxPosX = el.getPx() + el.getVel() * dx;
        int prxPosY = el.getPy() + el.getVel() * dy;

        int col = convertePosicaoIndice(prxPosX);
        int lin = convertePosicaoIndice(prxPosY);

        int colLarg = convertePosicaoIndice(prxPosX +
                        el.getLargura() - el.getVel());
        int linAlt = convertePosicaoIndice(prxPosY +
                        el.getAltura() - el.getVel());

        if (foraDaGrade(col, lin) || foraDaGrade(colLarg, linAlt))
            return true;
```

Nossa validação de movimento não se restringe mais apenas aos objetos da classe `Pizza`, ele aceita qualquer objeto que for um `Elemento`, mas somente os `Legumes` podem cruzar a linha para entrar e sair da jaula. Logo, verificamos se o `Elemento` a ser validado é uma instância da classe `Legume`, e fazemos o *cast* para verificar se o objeto está no modo `PRESO`.

```
    // Validar linha branca
        if (el instanceof Legume) {
            if (grade[lin][col] == Nivel.LN ||
                grade[lin][colLarg] == Nivel.LN ||
                grade[linAlt][col] == Nivel.LN ||
                grade[linAlt][colLarg] == Nivel.LN) {

                if (el.isAtivo() ||
                    ((Legume) el).getModo() == Modo.PRESO)
```

```
                return false;

            return true;
        }
    }

    if (grade[lin][col] >= Nivel.BL ||
        grade[lin][colLarg] >= Nivel.BL ||
        grade[linAlt][col] >= Nivel.BL ||
        grade[linAlt][colLarg] >= Nivel.BL) {

        return false;
    }

    return true;
}
```

Antes, tínhamos um único valor para representar a parede do cenário, mas agora qualquer valor maior ou igual a `Nivel.BL` (que é -1) representa um obstáculo. Assim, agora temos diversos blocos e devemos considerar que um deles pode ser atravessado dependendo do estado do personagem.

Para simplificar as coisas, apenas marcamos a posição desse bloco na variável `Nivel.LN`, e verificamos primeiramente se algum `Legume` colidiu com esse bloco, já que somente os inimigos podem atravessar essa parede. Depois, verificamos se houve colisão com qualquer outro bloco.

Por fim, nossa `grade` agora é uma cópia de `Nivel.cenario` para que, ao carregar novamente o jogo, todas as pastilhas que já foram comidas voltem ao seu lugar.

```
...
grade = copiaNivel(Nivel.cenario);
...
private int[][] copiaNivel(int[][] cenario) {
    int[][] temp = new int[cenario.length][cenario[0].length];
    for (int lin = 0; lin < cenario.length; lin++) {
        for (int col = 0; col < cenario[0].length; col++) {
            temp[lin][col] = cenario[lin][col];
        }
```

```
    }

    return temp;
}
```

Apenas copiamos os valores de um array para o outro, trabalhando em nova instância para não modificar o array original. Se você ficou com a vontade de criar seus próprios cenários e imagens (e eu realmente espero que tenha ficado), ou simplesmente queira modificar os arquivos do jogo, você pode usar o Gimp (http://www.gimp.org) , que é um editor de imagens gratuito.

> **Dica**
>
> Com muito esforço, Pixel Art é o tipo de desenho que eu consigo fazer. Por sorte, existe muito material na internet, como a comunidade Pixel Join (conteúdo em inglês): http://pixeljoint.com/.

7.3 Codificando o jogo

Nosso objeto `ImageIcon` é a grande novidade deste capítulo, e não perdemos tempo em usá-lo no método `desenhar`, chamando `drawImage` (com seus dez parâmetros) do objeto `Graphics2D` para isso.

Mal começamos a utilizar imagens e já trabalhamos com sprites e tiles, aplicando recortes, espelhando e sobrepondo as imagens, que não são poucas: `azeitona.png`, `olhos.png`, `pepperoni.png`, `sprite_inimigos.png`, `sprite_pizza.png` e `tiles_cenario.png`. E mesmo nosso cenário sendo composto por várias figuras, usamos o poder da matemática para, a partir do valor da grade, saber qual linha e coluna devemos desenhar.

Na primeira versão do jogo, trabalhamos apenas com um objeto que representava tanto o protagonista quanto os antagonistas, mas agora cada um deles é representado por sua própria classe, `Pizza.java` e `Legume.java`. Fizemos uma pequena revisão nos códigos alterados

da classe `JogoCenario.java`, e nenhuma mudança em nossas classes `Jogo.java` e `JogoCenario.java`.

Já fazia alguns capítulos que não expandíamos nossa biblioteca, então adicionamos em nossa classe `Elemento.java` melhorias para trabalhar com a direção do objeto e imagens. Confira em: https://github.com/logicadojogo/fontes/tree/master/Capo7.

Não fizemos aqui

- Aplicar rotação nas imagens;
- Apresentar os personagens na introdução do jogo.

Melhore você mesmo

- Aplique efeitos sonoros;
- Utilize suas próprias imagens para personalizer o jogo.

7.4 Resumo

Vimos como trabalhar com imagens, como é fácil carregar uma imagem simples e que segmentar por linhas e colunas pode ajudar na utilização de imagens complexas que representem múltiplas figuras. Espero que agora você consiga trabalhar com sprites e tiles sem ter de quebrar a cabeça.

Nossa manipulação de imagens está apenas no começo, mas muito mais pode ser feito com Java que não caberia neste livro. Até mesmo porque estamos nos caminhando para os últimos capítulos. Então, vamos nos focar no que for útil para os nossos jogos.

Capítulo 8

Um jogo de ângulos e rotações

Antes de falarmos de Asteroids, vamos falar da matemática envolvida nele. Isso, é claro, sem falar muito de matemática (ou pelo menos não tanto quanto seu professor gostaria), já que a classe `java.lang.Math` que o Java nos provê é cheia de recursos.

Começamos com efeitos de rotação, escalonamento e posicionamento, depois criamos um jogo chamado *Nave Quebrada*, para testarmos o que aprendemos. Prepare-se para:

- Rotação;
- Escalonamento;
- Transição;
- Ângulos.

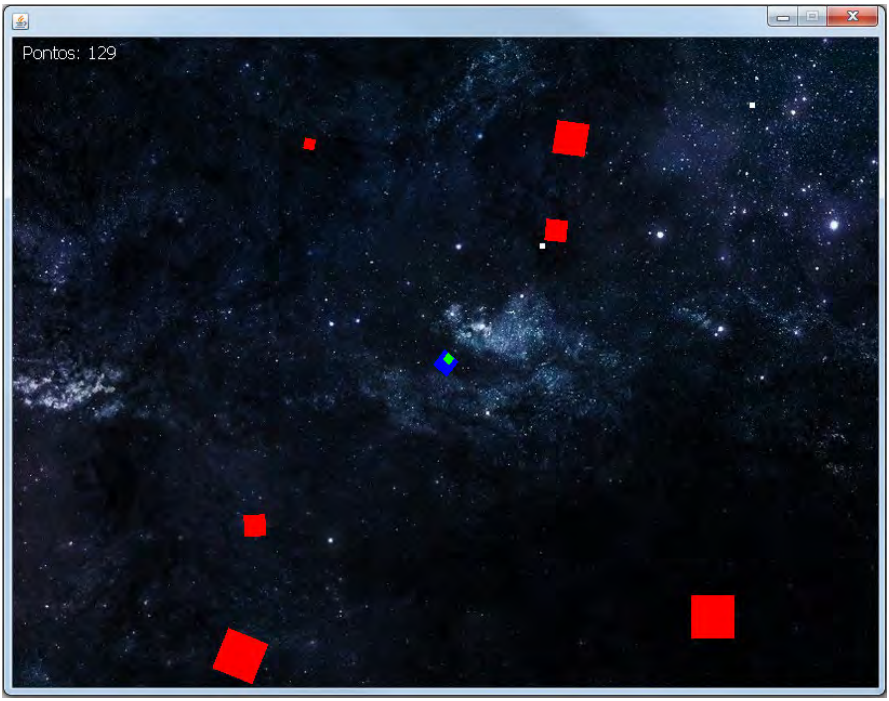

Fig. 8.1: Jogo Nave Quebrada

8.1 Rotação

A classe `Graphics2D` nos oferece dois métodos para rotação dos desenhos e imagens. O primeiro é `rotate(double theta)`, e o segundo `rotate(double theta, double x, double y)`, sendo que `theta` é o ângulo da rotação em radianos.

Como trabalharemos com valores em graus, utilizamos `Math.toRadians(graus)` para conversão. Começando com exemplos, vamos desenhar um quadrado cinza de fundo, aplicar uma rotação de 180°, e depois desenhar um quadrado menor na cor amarela, ambos no eixo 0.

Considere que as alterações no objeto `g2d` refletem em toda a nossa tela de pintura:

```
@Override
publicvoid paintComponent(Graphics g) {

    Graphics2D g2d = (Graphics2D) g;

    g2d.setColor(Color.GRAY);
    g2d.fillRect(0, 0, 200, 200);

    float anguloEmRadiano = (float) Math.toRadians(180);
    g2d.rotate(anguloEmRadiano);

    g2d.setColor(Color.YELLOW);
    g2d.fillRect(0, 0, 40, 40);
}
```

O quadrado cinza não será rotacionado, uma vez que foi desenhado antes da chamada do método `rotate`. E o quadrado amarelo acabará fora da tela, isso porque a rotação foi feita com base nas coordenadas o dos eixos X e Y da tela:

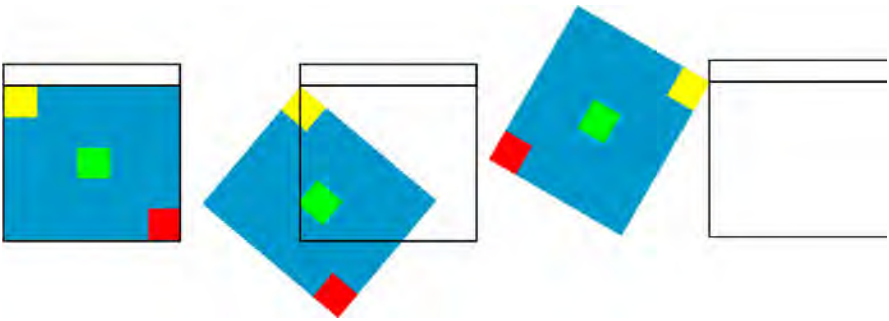

Fig. 8.2: Giro padrão

A figura anterior exemplifica mais ou menos onde o desenho foi parar ao usar uma rotação de 90°. Mas é claro que temos maneiras de fazer uma rotação levando como base o eixo central, uma delas é reposicionar nossa tela antes de aplicar a rotação e depois voltá-la a posição original.

Para alterar a posição da tela, usamos o método `translate`, que recebe como parâmetro o eixo X e Y da nova posição. A tela deve ser posicionada

de forma que suas coordenadas `0x` e `0y` fiquem no centro da janela antes e depois da rotação. Para isso, usamos metade da largura e altura da tela como parâmetros, que neste caso têm o mesmo valor de `TAMANHO_TELA`:

```
int meio = TAMANHO_TELA / 2;
float anguloEmRadiano = (float) Math.toRadians(180);

g2d.translate(meio, meio);
g2d.rotate(anguloEmRadiano);
g2d.translate(-meio, -meio);

g2d.setColor(Color.YELLOW);
g2d.fillRect(0, 0, 40, 40);
```

Primeiro, reposicionamos a tela deixando as coordenadas (`0, 0`) ao centro, depois efetuamos a rotação e, então, voltamos a tela para o ponto de origem. Assim, nosso quadrado amarelo será desenhado como se estivesse girando em torno da tela, mesmo sem alterar a posição do desenho no método `fillRect`, algo parecido com:

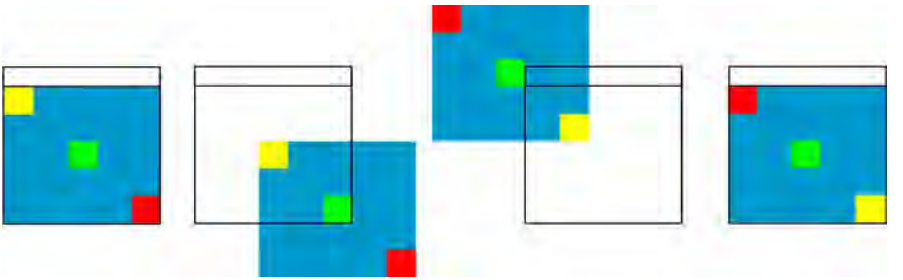

Fig. 8.3: Giro central

Inicialmente, temos a tela na posição original com um quadrado amarelo no canto superior esquerdo. A tela é reposicionada com `translate`, e a ponta do quadrado amarelo fica posicionada ao centro, assim podemos aplicar a rotação.

Depois, reposicionamos novamente a tela, subtraindo a metade do tamanho dela nos eixos X e Y. Como fazemos isso de uma só vez, visualizamos

apenas o desenho final, mas veremos como fazer isso de forma que vire uma animação.

> **Dica**
>
> Uma analogia seria você pensar em um pintor engessado da cabeça aos pés que só consegue mexer a mão, limitado a desenhar sempre na mesma posição. Então, para termos aquele mesmo desenho inclinado, em vez de pedir para o pintor se mover, você gira a tela de pintura e, quando ele acabar, você volta a tela para a posição original.

Observe que primeiro aplicamos a alteração na tela e, depois, pintamos o desenho, executando duas chamadas ao método `translate`. Para facilitar as coisas, a segunda chamada do método `rotate` nos poupa desse trabalho de reposicionamento:

```
int meio = TAMANHO_TELA / 2;
float anguloEmRadiano = (float) Math.toRadians(180);

g2d.rotate(anguloEmRadiano, meio, meio);

g2d.setColor(Color.YELLOW);
g2d.fillRect(0, 0, 40, 40);
```

Realmente saber como isso funciona e toda a matemática envolvida não é necessário para qualquer capítulo deste livro, mas ter a noção de como esses dois métodos se comportam ajudará muito no desenvolvimento deste e de qualquer outro jogo em que você queira girar objetos pela tela. Por isso exemplificamos, mesmo que superficialmente.

Até agora, nós aplicamos a rotação na tela toda, mas e se quisermos rotacionar os elementos individualmente? São os mesmos conceitos, só que levando em consideração o tamanho e a posição do elemento.

```
final int TAMANHO_TELA = 250;
int px = 10;
int py = 0;
```

```
int metadeLargEl = 200 / 2;
int metadeAltEl = 220 / 2;

g2d.rotate(anguloEmRadiano, metadeLargEl + px, metadeAltEl + py);
g2d.setColor(Color.YELLOW);
g2d.fillRect(px, px, larg, alt);
```

Nosso quadrado amarelo agora tem as dimensões 200 por 220. Sendo que `metadeLargEl` é metade da largura (`metadeAltEl` metade da altura) do elemento, que está posicionado em 10x e 0y. A posição do elemento deve ser levada em consideração, não somente a largura e altura. Como estamos somando aos eixos metade da largura e altura, o resultado será o quadrado amarelo girando em seu eixo central.

Mas a grande questão vem quando temos dois ou mais objetos na tela, já que após a chamada do método `rotate` (ou de outro método que transforme a tela) as alterações ou desenhos que vierem em seguida levam em conta os valores aplicados anteriormente, fazendo com que a ordem em que são aplicados gere resultados diferentes.

É agora que entra em cena a classe `java.awt.geom.AffineTransform`, representada pela variável `af`. Sem desmerecê-la, ela resumidamente serve como um recipiente para aplicarmos as transformadas.

```
int px = 0;
int py = 0;
int largEl = 200;
int altEl = 200;
int metadeLargEl = largEl / 2;
int metadeAltEl = altEl / 2;

AffineTransform af = g2d.getTransform();
```

Obtemos uma instância do objeto `AffineTransform` do próprio objeto `Graphics2D`, mas essa é só uma das várias formas de obtê-la. Faremos desta vez três desenhos, quadrados da mesma largura e altura (200, 200), nas cores amarelo, verde e vermelho, posicionados no topo ao lado esquerdo, centro e na base ao lado direito da tela, respectivamente.

```
// Quadrado Amarelo
g2d.rotate(anguloEmRadiano, metadeLargEl + px, metadeAltEl + py);
g2d.setColor(Color.YELLOW);
g2d.fillRect(px, px, largEl, altEl);
```

Até aqui tudo certo, desenhamos nosso quadrado amarelo no topo da tela aplicando uma rotação. Mas, antes de desenharmos o quadrado verde, precisamos retirar as transformações aplicadas, voltando a tela para a posição original. Logo, chamamos `setTransform` passando uma transformada "vazia", obtida no início do método.

```
// Quadrado Verde
g2d.setTransform(af);
px = TAMANHO_TELA / 2 - metadeLargEl;
py = TAMANHO_TELA / 2 - metadeLargEl;

g2d.rotate(anguloEmRadiano, metadeLargEl + px, metadeAltEl + py);
g2d.setColor(Color.GREEN);
g2d.fillRect(px, py, largEl, altEl);

// Quadrado Vermelho
g2d.setTransform(af);
px = TAMANHO_TELA - largEl;
py = TAMANHO_TELA - altEl;

g2d.translate(metadeLargEl + px, metadeAltEl + py);
g2d.rotate(anguloEmRadiano);
g2d.translate(-metadeLargEl - px, -metadeAltEl - py);

g2d.setColor(Color.RED);
g2d.fillRect(px, py, largEl, altEl);
```

As variáveis `px` e `py` agora representam o centro da tela, então desenhamos nosso quadrado verde. Fazemos o mesmo para o quadrado vermelho, mas o posicionamos no lado inferior direito, e utilizamos o primeiro exemplo do método `rotate`.

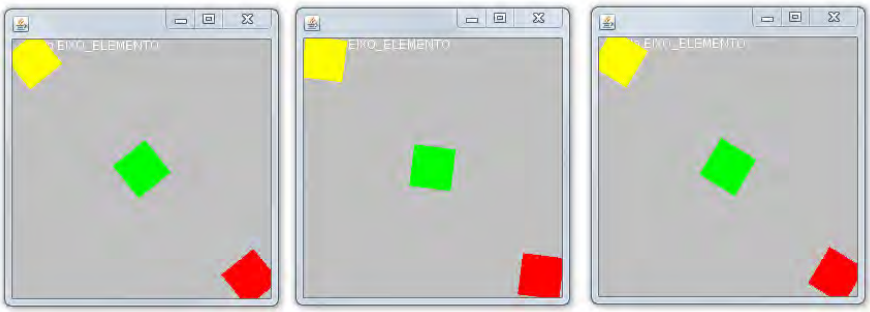

Fig. 8.4: Teste rotação no eixo central do desenho

Como o `AffineTransform` serve de recipiente, nós não precisamos aplicar as alterações direto no `Graphics2D`, por exemplo:

```
AffineTransform af = new AffineTransform();
af.rotate(anguloEmRadiano, metadeLargEl + px, metadeAltEl + py);

g2d.setTransform(af);
g2d.setColor(Color.YELLOW);
g2d.fillRect(px, px, largEl, altEl);
```

Aplicamos a rotação no objeto `af`, e depois usamos `g2d.setTransform` para aplicarmos o efeito em nosso desenho. Poderíamos utilizar somente o recipiente:

```
af.setToIdentity();
af.translate(metadeLargEl + px, metadeAltEl + py);
af.rotate(anguloEmRadiano);
af.translate(-metadeLargEl - px, -metadeAltEl - py);

g2d.setTransform(af);
g2d.setColor(Color.RED);
g2d.fillRect(px, py, largEl, altEl);
```

Este código tem o mesmo efeito do código anteriormente usado para desenhar o quadrado vermelho, mas aplicamos as transformações na nossa

própria instância do `AffineTransform`, e não diretamente no objeto `Graphics2D`. Importante notar que, depois que usamos `af` uma vez, voltamos a transformação ao estado original chamando o método `setToIdentity()`.

Vimos algumas maneiras de trabalhar com essas duas classes cheias de recursos, e ainda existem outras formas e diversos métodos que não citamos. Principalmente na classe `AffineTransform`, que engloba diversos métodos para aplicarmos transformação em nossas imagens, inclusive 4 versões do método `rotate` (sem contar o `quadrantRotate`).

Falaremos mais ao longo do capítulo, mas não cobriremos tudo, já que fugiríamos do escopo do livro. Recomendo que veja a documentação para conhecer mais.

Recomendo também que dê uma olhada na classe `RotacaoTeste.java` e teste o código. Faça alterações e veja coisas estranhas acontecendo quando não apagamos as transformações anteriores antes de aplicarmos uma nova.

8.2 Escalonamento e transição

Depois de tanto código e metáforas para rotacionar coisas no Java, falar de escalonamento e transição não é nenhum carnaval. Começando pela transição com o método `translate`, tanto da classe `Graphics2D` quanto da `AffineTransform`.

Falando de forma resumida e incompleta, ele reposiciona a tela nas coordenadas X e Y, e tudo o que for desenhado depois levará em consideração essas coordenadas. Por exemplo, o código a seguir desenha um quadrado amarelo em 25x e 25y:

```
g2d.translate(15, 25);
g2d.setColor(Color.YELLOW);
g2d.fillRect(10, 0, larg, alt);
```

Movemos a tela 15 pixels para a direita e 25 pixels para baixo, então as coordenadas iniciais não são mais (0,0). Ao preencher o quadrado na posição (10,0), a tela vai levar em consideração sua nova posição, que agora é (15,25). Portanto, o quadrado será desenhado no eixo 25x (15 + 10) e 25y (25 + 0).

Escalonar é uma forma de redimensionar o desenho ou imagem, e conseguimos isso usando o método `scale`, também disponível em ambas as classes. Para desenhar nosso quadrado com o dobro do tamanho, podemos fazer da seguinte forma:

```
g2d.scale(2, 2);
g2d.setColor(Color.YELLOW);
g2d.fillRect(0, 0, 40, 40);
```

Nosso quadrado vai ser desenhado 2 vezes maior na largura e altura, logo, mesmo que seu tamanho seja 40 por 40, na tela aparecerá com 80px. Também podemos diminuir ao invés de aumentar:

```
g2d.scale(0.5, 0.5);
g2d.setColor(Color.YELLOW);
g2d.fillRect(0, 0, 40, 40);
```

Agora o quadrado amarelo terá 50% do tamanho original, 20 por 20. Você pode passar valores diferentes para largura e altura, por exemplo `g.scale(2, 0.5)`, desde que os valores sejam diferentes de 0. Sendo iguais a 1, o resultado será o tamanho original.

Dica

A maior vantagem em utilizar `scale` e `translate` está quando aplicamos no jogo como um todo, podendo criar inúmeros efeitos, enquanto aplicar em elementos isoladamente pode causar erros. Um exemplo seria um elemento que está em uma posição nos eixos X e Y que colida com outro, mas que acabe sendo desenhado em outra parte da tela, atrapalhando o jogador.

A dica é: sempre pense na detecção de colisão ao aumentar ou diminuir o tamanho dos seus elementos.

Na classe `RotacaoTeste.java`, temos um modo que, além de girar, aumenta e diminui o tamanho do desenho:

```
int metadeLargEl = largEl / 2;
int metadeAltEl = altEl / 2;

// g2d.scale(escala, escala); // Chamada A

g2d.translate(TAMANHO_TELA / 2, TAMANHO_TELA / 2);
g2d.rotate(anguloEmRadiano);

g2d.scale(escala, escala); // Chamada B

g2d.translate(-metadeLargEl, -metadeAltEl);

// g2d.scale(escala, escala); // Chamada C

g2d.setColor(Color.YELLOW);
g2d.fillRect(0, 0, largEl, altEl);
```

A ordem em que aplicamos a transformação no desenho influencia diretamente o resultado. Para demonstrar isso, existem três chamadas para o método `scale` (chamada A, B e C), então, se comentar a linha da chamada B e descomentar a da chamada A ou C, você terá animações distintas.

Fig. 8.5: Girar e escalonar

Por ser apenas uma classe de teste, mantivemos as coisas bem simples e não vamos nos aprofundar em seus códigos. Porém, vale comentar que, para criar as animações da classe de teste, nossa rotação vai de 0 a 360 graus, e aumentamos até 9 vezes e depois diminuímos até 10% o tamanho do desenho

original.

```
If (System.currentTimeMillis() >= prxAtualizacao) {
    graus += inc;
    if (graus > 360)
        graus = 0;
    else if (graus < 0)
        graus = 360;

    escala += incEscala;
    if (escala > 9f || escala < 0.1f)
        incEscala = -incEscala;
    tela.repaint();
    prxAtualizacao = System.currentTimeMillis() + FPS;
}
```

Terminamos a introdução, e nada melhor que criar um jogo para sanar qualquer dúvida. Caso queira ver mais exemplos, acesse http://abrindoojogo.com.br, e procure por `AffineTransform` ou simplesmente Java.

8.3 Um jogo para girar e atirar

O jogo *Nave Quebrada* é sobre um piloto que teve sua nave danificada e não consegue mais sair do lugar, mas por sorte ainda pode girar e atirar. Convenientemente, ele parou no centro da tela, mas inimigos vermelhos aparecem de todos os lados, girando e indo em sua direção em diferentes velocidades.

> **Nota**
>
> Até a data de publicação deste livro, o jogo que remete a Asteroids mais recente chama-se *Galak-Z: The Dimensional*, criado pela 17-BIT (http://17-bit.com) .
>
> Para grandes coberturas sobre este e outros jogos, recomento o site independente *Overloadr* (http://overloadr.com.br) , que abrange amplamente o mundo dos videogames.

Toda vez que um inimigo acerta a nave, ela troca de cor e diminui seu tamanho (2 pixels a menos sem usar `scale`), a ponto que fique tão pequena que o jogo acabe. Embora usemos uma imagem de fundo, os elementos do jogo serão todos desenhados.

Teremos 3 elementos no jogo: `Asteroide`, `Nave` e `Tiro`. Além da classe `AffineTransform`, utilizaremos bastante a classe `java.lang.Math` e, embora a maioria dos métodos da classe `Math` trabalhem com valores em radianos e tipos `double`, para facilitar nosso lado, trabalharemos com ângulos de 0 a 360° e tipos `float`. Isso porque não precisamos de tanta precisão e ainda consumimos menos memória.

Aplicaremos rotação na nave e em nossos asteroides, isso quer dizer que usaremos transformações em cada elemento individualmente. Para uma alteração não sobrescrever outra, utilizamos `afAnterior` para guardar o valor original e depois restaurá-lo. Para as alterações individuais e temporárias, usamos a variável `af`. Ambas as variáveis são estáticas e obtidas por meio da nossa classe `Elemento`.

```
...
public class Elemento {

    public static AffineTransform afAnterior;
    protected static final AffineTransform af =
                                        new AffineTransform();
...
```

Outra mudança importante na classe `Elemento` é que, para termos uma movimentação mais suave, nossas variáveis `px` e `py` serão transformadas de inteiros para flutuantes (`int` para `float`).

```
private float px;
private float py;
public void setPx(float px) {
    this.px = px;
}
public void setPy(float py) {
    this.py = py;
}
```

Essa mudança não faz diferença na hora de definir o valor, apenas para obtê-lo, então mantivemos o retorno compatível com a versão anterior, fazendo o *cast* (convertendo) de `float` para `int`, e criamos métodos novos para retornar o valor como `float`.

```
public int getPx() {
    return (int) px;
}
public int getPy() {
    return (int) py;
}
public float getMovPx() {
    return px;
}
public float getMovPy() {
    return py;
}
```

O motivo disso é que os métodos de desenho do Java utilizam medidas em pixel e, por consequência, valores inteiros. Mas os cálculos que usamos na movimentação podem resultar em valores de ponto flutuante. Dessa forma, agradamos tanto a classe `Math` quanto a classe `Graphics2D`.

Nossa classe `Nave.java` tem apenas duas variáveis: uma para controlar o ângulo usado na rotação (variável `angulo`), e `inverteCor` para indicar quando a nave for atingida:

```
public class Nave extends Elemento {

    private float angulo;
    private boolean inverteCor;
...
    public float getAngulo() {
        return angulo;
    }

    public void setAngulo(float angulo) {
        this.angulo = angulo;
    }
```

```
    public void inverteCor() {
        inverteCor = !inverteCor;
    }
}
```

É no método `desenha` que as coisas ficam mais interessantes. Entretanto, antes de desenharmos qualquer coisa, guardamos as transformações atuais, já que precisamos manter o mesmo estado para que os próximos elementos sejam desenhados corretamente.

```
@Override
public void desenha(Graphics2D g) {
    afAnterior = g.getTransform();

    float rad = (float) Math.toRadians(getAngulo());
    af.setToIdentity();
    af.rotate(rad, getLargura() / 2 + getPx(),
                   getAltura() / 2 + getPy());
    g.setTransform(af);
```

Depois, convertemos nosso ângulo em graus para radianos, para rotacionarmos nosso desenho. Como a variável `af` é compartilhada com outros elementos, resetamo-la com o método `setToIdentity()`. Neste caso, estamos ignorando qualquer transformada anterior.

```
if (inverteCor)
        g.setColor(Color.BLUE);
    else
        g.setColor(Color.YELLOW);

    g.fillRect(getPx(), getPy(), getLargura(), getAltura());

    g.setColor(Color.GREEN);
    g.fillRect(getPx() + getLargura() / 2,
          getPy() + getAltura() / 4,
          getLargura() / 2, getAltura() / 2);

    g.setTransform(afAnterior);
}
```

Inicialmente, nossa nave é amarela, mas se estiver com a cor invertida, será desenhada na cor azul. Desenhamos um quadrado menor na cor verde para indicar a origem do disparo. Depois, restauramos a transformação anterior, obtida no começo do método.

Fig. 8.6: Nave antes e depois da colisão

Como o ângulo inicial é 0º (ou 360º), consideramos como frente da nave o lado direito do quadrado, onde posicionamos um segundo quadrado verde com metade da largura e altura do primeiro, de forma centralizada.

> **Dica**
>
> Estamos apagando as transformações com `setToIdentity()`, mas também podemos "acumular" transformações usando `af.setTransform( afAnterior )`.

Nossa classe `Asteroide` é um quadrado vermelho que não inverte a cor, mas tem rotação constante, gerenciada pela variável `anguloRotacao`, que é incrementada constantemente.

```
public class Asteroide extends Elemento {

    private float angulo;
    private float anguloRotacao;

    @Override
    public void desenha(Graphics2D g) {
        if (!isAtivo())
            return;

        afAnterior = g.getTransform();
```

```
        af.setToIdentity();
        af.rotate(anguloRotacao,
             getLargura() / 2 + getPx(),
             getAltura() / 2 + getPy());
        g.setTransform(af);

        g.setColor(Color.RED);
        g.fillRect(getPx(), getPy(),
              getLargura(), getAltura());
        g.setTransform(afAnterior);
    }
```

A grande novidade dessa classe (e desse jogo) está no método `atualiza`. Nós simplesmente não utilizamos variáveis `dx` e `dy` para controlar a direção do elemento nos eixos X e Y, como nos jogos anteriores. Em vez disso, calculamos a direção com base no ângulo.

```
@Override
public void atualiza() {
    if (!isAtivo())
        return;

    float cos = (float) Math.cos(
                Math.toRadians(angulo));
    float sen = (float) Math.sin(
                Math.toRadians(angulo));

    setPx(getMovPx() + cos * getVel());
    setPy(getMovPy() + sen * getVel());

    anguloRotacao++;
    if (anguloRotacao == 360)
        anguloRotacao = 0;
}
```

Para controlar a direção em que o asteroide se move, usamos uma forma mais sofisticada, que é por meio do **seno** e **cosseno** com base no ângulo, que pode ser entre o asteroide e a nave do jogador, por exemplo.

Usamos a função `Math.sin` para calcular o seno, e `Math.cos` para o cosseno. Ambas as funções pedem o ângulo em radianos, então fazemos a conversão na chamada dos métodos. Uma vez com os valores de `cos` e `sen`, definimos a nova posição somando a posição atual (em ponto flutuante) multiplicada pela velocidade do elemento.

Podemos obter o ângulo entre os dois elementos (com uma função que veremos mais adiante), assim calculamos o cosseno e seno para saber o quanto devemos nos mover nos eixos XY (cosseno para o eixo X, e seno para o eixo Y) para chegarmos ao destino, fazendo assim os asteroides irem na direção desejada, independente de suas localizações na tela.

Usamos uma velocidade de movimento variada para cada asteroide, e `anguloRotacao` para girá-lo rapidamente (sem convertê-lo em radianos). Eles não têm muita criatividade para girar, mas têm um efeito de *shuriken* e são destruídos por tiros. Veja nossa classe mais simples:

```
public class Tiro extends Elemento {
    private float angulo;

    ...

    @Override
    public void atualiza() {
        if (!isAtivo())
            return;

        float cos = (float) Math.cos(
                    Math.toRadians(angulo));
        float sen = (float) Math.sin(
                    Math.toRadians(angulo));

        setPx(getMovPx() + cos * getVel());
        setPy(getMovPy() + sen * getVel());
    }
...
}
```

A classe `Tiro` utiliza o método de desenho padrão (da classe `Elemento`), e sua única preocupação é saber em qual ângulo o jogador es-

tava quando efetuou o disparo, que segue em única direção até encontrar um asteroide ou sair da tela – em ambos os casos, ficando inativo.

Vimos que o ângulo é a chave da rotação e movimentação dos elementos desse jogo, então precisamos obtê-lo corretamente. Para isso, pedimos mais uma vez ajuda do Java e sua classe `Math`, conforme veremos a seguir.

8.4 Jogando o código

Nossa classe `JogoCenario.java`, onde começamos o jogo com uma nave, vinte e cinco tiros e cinquenta aerólitos (isso mesmo, não são asteroides, são aerólitos).

```
private Nave nave;
private Tiro[] tiros = new Tiro[25];
private Asteroide[] aerolitos = new Asteroide[50];

private Texto texto = new Texto();
private Random rand = new Random();
private Estado estado = Estado.JOGANDO;

private int pontos;
private int adiciona = 2;
private int contadorTiro;
private int intervalo = 60;
private int temporizador = 0;

private float graus;
```

Usamos a variável `adiciona` para saber quantos inimigos adicionamos na tela a cada `intervalo` de 60 atualizações ou 3 segundos. Vale lembrar que definimos nosso `FPS` com 20 atualizações por segundo. A variável `contadorTiro` é usada como índice do array `tiros`.

Todos os objetos são instanciados no método `carregar`, e não dinamicamente durante o jogo conforme as interações forem ocorrendo. Isso poderia ser feito, mas criar uma nova instância terá um custo operacional maior do que já ter o objeto e apenas ativá-lo (lembrando de que estamos falando de milissegundos).

```
@Override
public void carregar() {

    texto.setCor(Color.WHITE);

    for (int i = 0; i < tiros.length; i++) {
        tiros[i] = new Tiro(5, 5);
        tiros[i].setVel(5);
    }

    for (int i = 0; i < aerolitos.length; i++) {
        aerolitos[i] = new Asteroide();
    }

    nave = new Nave(0, 0, 40, 40);
    nave.setAtivo(true);
    nave.setCor(Color.YELLOW);

    Util.centraliza(nave, largura, altura);
}
```

Dessa forma, além de reaproveitarmos os objetos, deixamos claro o quanto de recursos precisamos, diminuindo o risco de o jogo ficar lento com o passar do tempo.

Dica

Não pude deixar de testar o jogo com grandes quantidades de elementos na tela, então aumentei a quantidade de tiros para 2.500, com 5.000 asteroides sendo adicionados 25 por vez e não tive sensação de lentidão ou travamento. O efeito ficou legal:

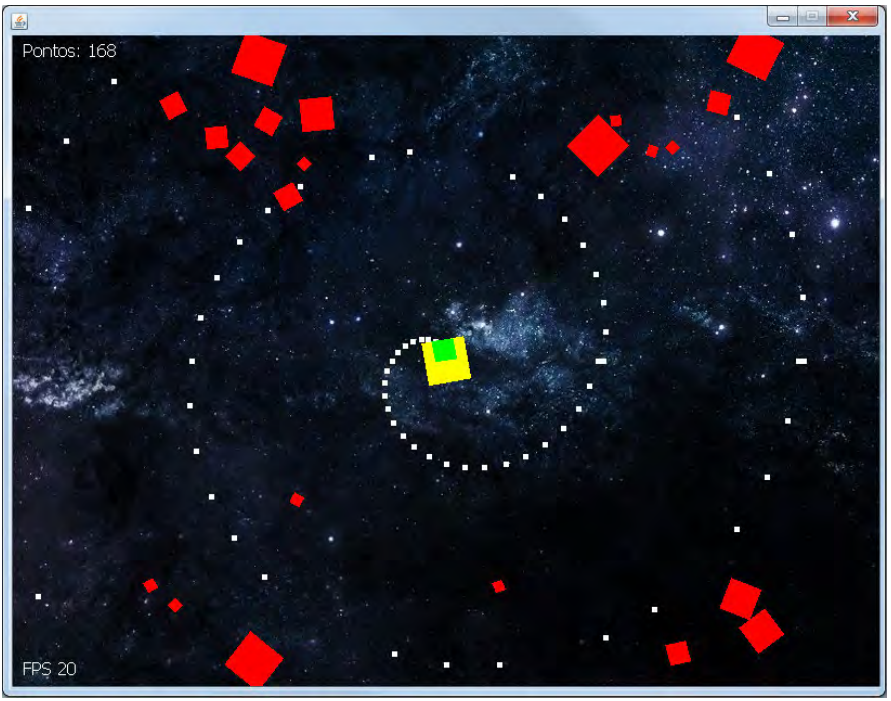

Fig. 8.7: Tiros, muitos tiros

A vida do jogador é o tamanho da própria nave que diminui conforme é atingida, e ele perde se ela ficar muito pequena. Para dar chance ao jogador, os inimigos são adicionados em quantidades e intervalos pré-definidos pelas variáveis `adiciona` e `intervalo`.

```
@Override
public void atualizar() {

    if (estado != Estado.JOGANDO) {
        return;
    }

    if (nave.getLargura() < 5) {
        estado = Estado.PERDEU;
        return;
    }

    if (temporizador == intervalo) {
        temporizador = 0;
        maisAerolitos();

    } else
        temporizador++;
```

O tamanho mínimo do jogador é 5 pixels, e novos aerólitos são adicionados ao jogo quando o `temporizador` se iguala ao `intervalo`. A nave não se movimenta pela tela, mas quando o jogador pressiona para esquerda ou direita para girá-la, diminuímos ou aumentamos os graus da nave sem extrapolar nosso range.

```
if (Jogo.controleTecla[Jogo.Tecla.ESQUERDA.ordinal()])
    graus -= 10;
else if (Jogo.controleTecla[Jogo.Tecla.DIREITA.ordinal()])
    graus += 10;

if (graus < 0)
    graus += 360;
else if (graus > 360)
    graus -= 360;

nave.setAngulo(graus);
```

Ao pressionar o botão de ação (mapeado para a tecla espaço), chamamos o método `adicionarTiro` que coloca novos tiros na tela. Para saber

para onde ela está apontada, levamos em consideração seu ângulo, que usa o mesmo ângulo para direcionar o tiro.

```
if (Jogo.controleTecla[Jogo.Tecla.BC.ordinal()]) {

    adicionarTiro(nave.getAngulo());
    Jogo.liberaTecla(Jogo.Tecla.BC);
}
```

Nossos aerólitos começam na parte superior ou inferior da tela, mas inicialmente fora dela, e eles vagam aleatoriamente ou em direção a nave. Quando eles colidem com o tiro ou com a nave, ou simplesmente cruzam para fora da tela além do ponto de origem, são destruídos (inativados).

```
for (Asteroide ast : aerolitos) {

    if (!ast.isAtivo())
        continue;

    // Asteroides começam fora da tela
    if (ast.getPy() < ast.getAltura() * -2 ||
                ast.getPy() > ast.getAltura() + altura) {
        ast.setAtivo(false);
        continue;
    }

    if (ast.getPx() + ast.getLargura() < 0 ||
                ast.getPx() > largura) {
        ast.setAtivo(false);
        continue;
    }
```

Para saber se estão fora da área permitida, verificamos se a posição do objeto está além ou aquém da posição inicial do elemento, com um pouco mais de margem no eixo Y. O jogador ganha pontos quando o asteroide é atingido por um tiro de acordo com a velocidade dele, então asteroides mais rápidos dão mais pontos.

```
for (Tiro tiro : tiros) {
        if (Util.colide(ast, tiro)) {
```

```
            ast.setAtivo(false);
            tiro.setAtivo(false);
            pontos += ast.getVel();
            break;
        }
    }

    if (Util.colide(ast, nave)) {
        ast.setAtivo(false);
        nave.setLargura(nave.getLargura() - 2);
        nave.setAltura(nave.getAltura() - 2);
        nave.inverteCor();

        Util.centraliza(nave, largura, altura);
        continue;
    }

    ast.atualiza();
}
```

Quando o asteroide atinge a nave, ela fica dois pixels menor na largura e altura, e tem sua cor invertida. Como seu tamanho foi alterado, precisamos centralizá-la na tela novamente. Por fim, verificamos se os tiros remanescentes (que estão ativos) acabaram no espaço (saíram da tela) para serem inativados, ou atualizados.

```
for (Tiro tiro : tiros) {
    if (!tiro.isAtivo())
        continue;

    if (Util.saiu(tiro, largura, altura))
        tiro.setAtivo(false);
    else
        tiro.atualiza();
}
```

Os dois métodos auxiliares que veremos a seguir, `adicionarTiro` e `maisAerolitos`, são os responsáveis pelo desafio do jogo. Embora nosso

array de tiros tenha um tamanho fixo, não limitamos os disparos do jogador. Utilizamos o `contadorTiro` para ir do último até o primeiro índice do array, e então voltamos para o último.

Como estamos reaproveitando a mesma instância da classe `Tiro`, pode acontecer de o tiro sumir da tela antes de sair dela ou de colidir com um inimigo. Para evitar que isso aconteça, poderíamos percorrer o array em busca de um tiro inativo em vez de usarmos um contador (faremos algo similar com os `asteroides`).

O parâmetro `angulo` que recebemos é o que consideramos ser a frente da nave, onde desenhamos o quadrado verde. Para o disparo sair centralizado com a nave, usamos a posição da nave mais metade da sua largura menos a metade da largura do tiro para posicioná-lo no eixo X, o equivalente para o eixo Y. Isso é similar ao que fizemos para centralizar os elementos na tela.

```
private void maisAerolitos() {
    int contador = 0;
    for (int i = 0; i < aerolitos.length; i++) {
        if (contador == adiciona)
            break;

        Asteroide ast = aerolitos[i];

        if (ast.isAtivo())
            continue;

        contador++;
        ast.setAtivo(true);
```

A quantidade de aerólitos a serem adicionados depende do valor de `adiciona` (fixado em 2), mas pode acontecer de não adicionarmos nenhum inimigo caso todos estejam ativos.

Os asteroides possuem 5 tamanhos diferentes, de 10 até 50 pixels, e começam em qualquer ponto do eixo X. Mas, no eixo Y, ou começam na parte superior ou inferior da tela, resultado que obtemos multiplicando `rand.nextInt(2)` (que retorna 0 ou 1) pela `altura` da tela, subtraindo a altura do elemento se ele começar na parte superior.

```
ast.setAltura((rand.nextInt(4) + 1) * 10);
ast.setLargura(ast.getAltura());

ast.setPx(rand.nextInt(largura));

int py = rand.nextInt(2) * altura;
if (py == 0)
    py = py - ast.getAltura();

ast.setPy(py);
ast.setVel(rand.nextInt(3) + 1);
```

Outra variação ocorre na velocidade dos inimigos, que vai de 1 até 3 e influenciam na pontuação. Porém, a principal variação está na seleção do modo de jogo onde temos os famosos: fácil, normal e difícil.

```
switch (Jogo.nivel) {
case 0:
    // Modo Facil: vai para qualquer lado
    ast.setAngulo(rand.nextInt(360));
    break;

case 1:
    // Modo Normal: vai em ângulos próximos
    // a nave do jogador
    if (ast.getPy() <= 0)
        ast.setAngulo(90);
    else
        ast.setAngulo(270);
    break;
```

No primeiro modo, os aerólitos vão em qualquer direção aleatoriamente e muitos passam longe do jogador. Já no segundo, eles vão em ângulos retos, tendo mais chances de atingir a nave. O modo difícil, também chamado de `Teleguiado`, não tem esse nome à toa, já que todos os inimigos vão na direção do jogador.

Como eles têm posições aleatórias, precisamos calcular o ângulo com base na posição do aerólito e da nave, obtida com `Math.atan2`.

```
default:
            // Modo Dificil: todos vão em direção a
            // nave do jogador
            float arco = (float) Math.atan2(
                            nave.getPy() - ast.getPy(),
                            nave.getPx() - ast.getPx());

            float angulo = (float) Math.toDegrees(arco);
            ast.setAngulo(angulo);
            break;
        }
    }
}
```

Passamos dois parâmetros para a função `Math.atan2`, ambos envolvendo a posição dos nossos objetos, sendo eles o destino subtraído pela origem. Neste caso, o destino é nosso objeto `nave`, e a origem, nosso asteroide `ast`.

Observe que passamos no primeiro parâmetro o eixo Y, ao contrário da maioria das funções que utilizamos, fora isso, basta lembrar de que subtraímos a posição para onde queremos ir da posição onde está o objeto que vai se mover, a não ser que se queira ir para o lado oposto, útil em algoritmos de fuga.

Como estamos trabalhando com ângulos em graus, convertemos o resultado com `Math.toDegrees` e, depois, na hora de movimentar o elemento dentro do seu método `atualiza`, convertemos novamente em radianos com `Math.toRadians`. Então, quando se sentir mais à vontade ao trabalhar com ângulos, prefira usar radianos para fazer menos conversões.

Vimos o método `Math.atan2` na prática, mas se precisar de um pouco de teoria, recomendo http://gamedev.stackexchange.com/questions/14602/what-are-atan-and-atan2-used-for-in-games (conteúdo em inglês).

8.5 Codificando o jogo

Primeiro, vimos como aplicar efeitos de rotação, escalonamento e transição aos nossos desenhos, utilizando as classes `Graphics2D` e

`AffineTransform`. Criamos alguns exemplos que você pode conferir na classe `RotacaoTeste.java`, em https://github.com/logicadojogo/fontes/tree/master/Capo8, dentro do pacote teste.

Nossa classe `Elemento` agora mantém uma instância de `AffineTransform` para aplicarmos efeitos em nossos novos objetos de forma temporária: `Asteroide`, `Nave` e `Tiro`. Antes de aplicarmos qualquer transformada, usamos `afAnterior` para manter as transformadas anteriores do objeto `Graphics2D`, mesmo que ainda não estejamos aplicando nenhum efeito (que veremos no próximo capítulo).

Não estamos utilizando as variáveis `dx` e `dy`, e as variáveis `px` e `py` agora são do tipo `float`, para termos uma movimentação mais fluida para nossos elementos que usam ângulos.

Utilizamos valores em graus de 0º a 360º, mas as principais funções matemáticas da classe `Math` que vimos trabalham com radianos. Assim, usamos `Math.toRadians` para converter graus em radianos, e `Math.toDegrees` para radianos em graus.

A lógica mais interessante desse jogo está em fazer os asteroides perseguirem a nave do jogador. Para isso, primeiro obtemos o ângulo correto entre os dois objetos com `Math.atan2`, depois passamos esse ângulo para `Math.cos` e `Math.sin`, que nos retornam o cosseno e seno, respectivamente. Por fim, usamos esses valores para atualizarmos a movimentação do nosso personagem, cosseno para o eixo X e seno para o eixo Y.

Temos um bom exemplo de como criar níveis diferentes de dificuldade para o jogador, que nomeamos como: *Sem radar* (fácil), *Com radar* (normal) e *Teleguiado* (difícil). Na primeira opção, o angulo é qualquer coisa entre 0º e 359º, então muitos passarão longe do jogador, podendo nem chegar a aparecer na tela. Na segunda opção, ao menos garantimos que os inimigos cruzarão a tela na vertical. Já a última é implacável, obtemos as coordenadas da nave e direcionamos todas as pedras espaciais para ela.

Não fizemos aqui

- Movimentar a nave pela tela;
- Controlar a nave com o mouse.

Melhore você mesmo

- Contar acertos seguidos para aumentar a pontuação do jogador;
- Formas de tiro que usem ângulos diferentes;
- Utilizar imagens.

8.6 Resumo

Vimos neste capítulo como aplicar transformadas em nossos desenhos e a matemática necessária para o jogo funcionar. Usaremos mais do que foi apresentado no próximo capítulo e, mesmo assim, ambos os tópicos são mais abrangentes do que conseguiremos cobrir.

Graças às transformações do desenho, nós conseguimos criar diversos efeitos dentro do próprio jogo, incluindo animações. Jogos que envolvam geometria e física ficam mais fáceis com as classes e métodos que o Java nos fornece, e posso dizer por mim que seno e cosseno nunca se fizeram tão úteis antes de utilizá-los em jogos.

Capítulo 9

Asteroids: o jogo que sabe representar a física

Lyle Rains deu a ideia, e Ed Logg a colocou em mais de 70 mil máquinas de jogos nos Estados Unidos em 1979, vendidas pela Atari. Porém, o número mais impressionante é o de um garoto que jogou Asteroids por 36 horas antes de cansar.

Utilizando gráficos vetoriais, o jogo se diferenciava da maioria dos jogos da época (que eram pixelizados), tendo velocidade e nitidez, além de guardar as iniciais (ou 3 letras quaisquer) dos jogadores que pontuassem mais e, é claro, era de uma física desafiadora.

Guardamos para este capítulo:

- Animação de Sprite;

- Efeito de aproximação (zoom);
- Melhorias no loop do jogo.

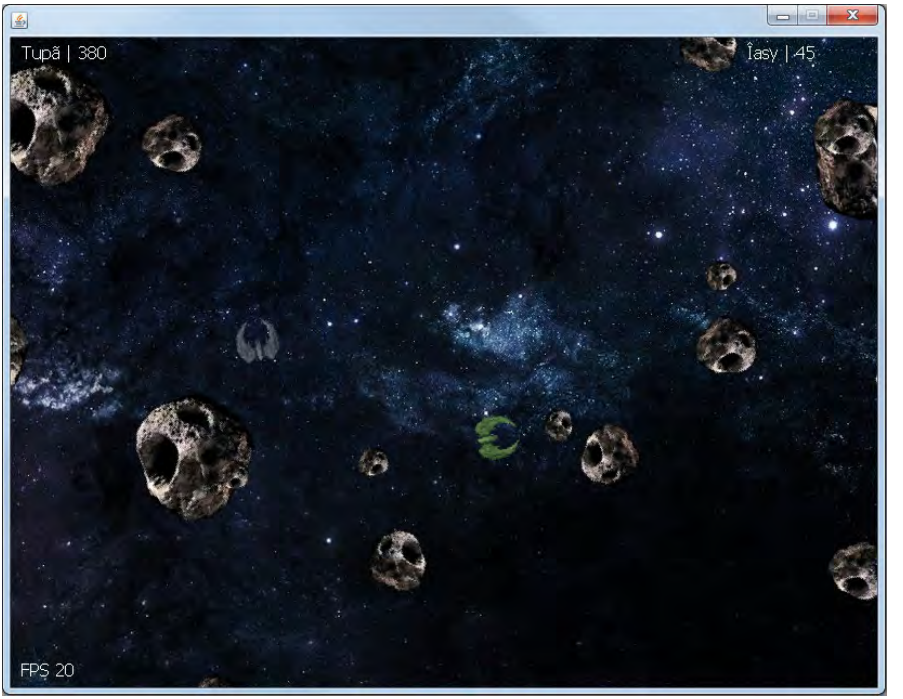

Fig. 9.1: Ângulos: o jogo

9.1 Um é bom, dois é multiplayer

Asteroids pode ser antigo, mas seus conceitos são usados em inúmeros jogos atuais, inclusive no jogo do capítulo anterior. A nossa versão deste capítulo também brincará bastante com rotações e ângulos, mas desta vez usando imagens, além da possibilidade de dois jogadores (modo multijogador ou multiplayer) – o primeiro controlado pelo teclado, e o segundo pelo mouse, disputando quem destrói mais (ou erra menos) asteroides.

Concentramos todas as imagens do jogo em objetos estáticos na classe `Recursos.java`:

```
public enum Imagem {
    FUNDO, JOGADOR_A, JOGADOR_B, TIRO_A, TIRO_B,
    AST_A, AST_B, AST_C, EXPLOSAO_AST, COLISAO_AST
}

private static final String DIR_IMG = "imagens/";

private static ImageIcon[]
        imagens = new ImageIcon[Imagem.values().length];
```

Centralizamos todas as imagens aqui, e quem precisar pede um `ImageIcon` utilizando o próprio `enum` da classe. Assim, se quisermos renomear ou substituir uma imagem (ou até mesmo refazer toda a lógica de carregamento), só precisamos alterar um arquivo.

```
public static void carregarImagens() {
    imagens[Imagem.FUNDO.ordinal()] =
                new ImageIcon(DIR_IMG + "fundo.png");
    imagens[Imagem.JOGADOR_A.ordinal()] =
                new ImageIcon(DIR_IMG + "nave_jogador_1.png");
    imagens[Imagem.JOGADOR_B.ordinal()] =
                new ImageIcon(DIR_IMG + "nave_jogador_2.png");
    imagens[Imagem.TIRO_A.ordinal()] =
                new ImageIcon(DIR_IMG + "tiro.png");
    imagens[Imagem.TIRO_B.ordinal()] =
                new ImageIcon(DIR_IMG + "tiro.png");
```

A maioria das imagens foi obtida neste grande repositório chamado internet, em uma busca por Espaço Sideral, Meteoritos e Battlestar para as naves dos jogadores, que apenas têm cor diferente. Nós mesmos fizemos os tiros e as explosões e, mesmo o tiro sendo o mesmo para os dois jogadores, nossa classe já está preparada para carregar imagens diferentes para cada um.

```
imagens[Imagem.AST_A.ordinal()] =
                new ImageIcon(DIR_IMG + "asteroide_grande.png");
    imagens[Imagem.AST_B.ordinal()] =
                new ImageIcon(DIR_IMG + "asteroide_medio.png");
    imagens[Imagem.AST_C.ordinal()] =
                new ImageIcon(DIR_IMG + "asteroide_pequeno.png");
```

```
        imagens[Imagem.EXPLOSAO_AST.ordinal()] =
                    new ImageIcon(DIR_IMG + "explosao_asteroid.png");
        imagens[Imagem.COLISAO_AST.ordinal()] =
                    new ImageIcon(DIR_IMG + "colisao_asteroid.png");

}

public static ImageIcon getImagem(Imagem img) {
    return imagens[img.ordinal()];
}
```

O método `carregarImagens` é chamado antes de qualquer cenário ser iniciado e carrega todas as imagens do jogo. Se fosse um jogo maior, com diversos cenários e compostos por muitas imagens, o ideal seria carregar as imagens específicas de cada cenário, e não todas de uma vez, mas este não é o caso.

Veremos algumas dessas imagens ao longo do capítulo, lembrando de que todas estão na pasta `imagens`, na raiz do projeto.

Nossa biblioteca de código aumentou, e toda a matemática necessária (vista no capítulo anterior) foi movida para a classe `MatUtil.java`:

```
public static float moveAnguloEmX(float angulo) {
    return (float) Math.cos(Math.toRadians(angulo));
}

public static float moveAnguloEmY(float angulo) {
    return (float) Math.sin(Math.toRadians(angulo));
}
```

Para movimentar os personagens com base em seus ângulos, usaremos `moveAnguloEmX` e `moveAnguloEmY`, que fazem a conversão para radianos e de `double` para `float`. Dessa forma, calculamos o ângulo em cada eixo separadamente, mas também temos um método para aplicar o movimento diretamente no `Elemento`:

```
public static void move(Elemento el, float angulo, int vel) {
    float cos = (float) Math.cos(Math.toRadians(angulo));
```

```
    float sen = (float) Math.sin(Math.toRadians(angulo));

    el.setPx(el.getMovPx() + cos * vel);
    el.setPy(el.getMovPy() + sen * vel);
}

public static float corrigeGraus(float graus) {
    if (graus < 0)
        graus += 360;
    else if (graus > 360)
        graus -= 360;

    return graus;
}
```

Como estamos convertendo nosso valor em graus para radianos, não precisamos nos preocupar se o valor passado foge do nosso range de 0º a 360º. Mesmo assim, utilizamos a função `corrigeGraus` para ter maior controle sobre ele.

No jogo anterior, calculamos a direção entre o asteroide e a nave, mas neste precisamos calcular a direção entre a nave do segundo jogador e o clique do mouse.

```
public static float calculaDirecao(int xDestino, int yDestino,
                    int xOrigem, int yOrigem) {
    return (float) Math.toDegrees(
                Math.atan2(yDestino - yOrigem,
                      xDestino - xOrigem) );
}

public static float calculaDirecao(Elemento origem,
                                   Elemento destino){
    return (float) Math.toDegrees(
            Math.atan2(destino.getPy() - origem.getPy(),
                  destino.getPx() - origem.getPx()) );
}
```

A primeira chamada do método `calculaDirecao` recebe quatro inteiros que representam os eixos X e Y do ponto de destino e de origem. Sendo

que o destino será o clique do mouse e a origem a posição da nave no momento do clique. A segunda chamada do método faz o mesmo, levando em consideração a posição entre dois elementos, não utilizada nessa versão no jogo.

Os elementos que compõem o jogo são os mesmos do jogo anterior (`Nave`, `Tiro` e `Asteroide`), mas com comportamentos diferentes. Além disso, implementamos um efeito de zoom usando `scale` e `translate`, que ocorrerá toda vez que as naves dos jogadores se chocarem.

Essas transformações são feitas antes de desenharmos os elementos, então, é preciso agregar as transformadas em vez de apagá-las. Exploramos três formas diferentes de fazermos essa agregação, e cada um dos três elementos utiliza uma.

Quando uma nave atinge um asteroide, o jogador ganha pontos, e quando ela atinge vários seguidos sem errar, o jogador ganha mais pontos ainda. Para saber qual nave efetuou o disparo, o `Tiro` guarda a referência do objeto `Nave`:

```
public class Tiro extends Elemento {

    private Nave nave;
    private float angulo;

    @Override
    public void atualiza() {
        if (!isAtivo())
            return;
        MatUtil.move(this, angulo, getVel());
    }
```

Para desenhar o tiro, usamos dois objetos `AffineTransform`: um para guardarmos a transformação do objeto `Graphics2D` antes de aplicarmos as nossas transformações (`afAnterior`), e outro para aplicarmos as transformações referentes ao tiro (`af`), isso depois de mesclamos a transformada anterior com `af.setTransform(afAnterior)`.

```
@Override
public void desenha(Graphics2D g) {
```

```
    if (!isAtivo())
        return;

    afAnterior = g.getTransform();
    af.setTransform(afAnterior);
             af.rotate(Math.toRadians(angulo),
         getLargura() / 2 + getPx(),
         getAltura() / 2 + getPy());
    af.translate(getPx(), getPy());

    g.setTransform(af);
    g.drawImage(getImagem().getImage(), 0, 0, null);
    g.setTransform(afAnterior);
}
```

Esse é o primeiro dos três exemplos e é a pior prática para este caso, já que usamos dois objetos `AffineTransform` – agora, você sabe o que não fazer. Nos três elementos, o método `setImagem` ajusta a largura e altura do elemento de acordo com a altura e largura da imagem:

```
@Override
    public void setImagem(ImageIcon img) {
        super.setImagem(img);
        super.setAltura(img.getIconHeight());
        super.setLargura(img.getIconWidth());
    }

    public float getAngulo() {
        return angulo;
    }

    public void setAngulo(float angulo) {
        this.angulo = angulo;
    }

    public Nave getNave() {
        return nave;
    }
```

```
    public void setNave(Nave nave) {
        this.nave = nave;
    }
}
```

Nosso `Asteroide` tem quatro tamanhos: grande, médio, pequeno e poeira. Ele começa no tamanho grande e, toda vez que colide com o tiro ou a nave, diminui (divide), até chegar ao tamanho poeira.

```
public class Asteroide extends Elemento {

    public enum Tamanho {
        GRANDE, MEDIO, PEQUENO, POEIRA;
    }

    private float angulo;
    private float anguloRotacao;
    private Tamanho tamanho = Tamanho.GRANDE;

    @Override
    public void atualiza() {
        if (!isAtivo())
            return;
        MatUtil.move(this, angulo, getVel());
            anguloRotacao = MatUtil.corrigeGraus(
                        anguloRotacao + getVel());
    }
```

Neste exemplo, usando somente um objeto `AffineTransform`:

```
@Override
    public void desenha(Graphics2D g) {
        if (!isAtivo())
            return;
        af.setTransform(g.getTransform());
        af.rotate(Math.toRadians(anguloRotacao),
                    getLargura() / 2 + getPx(),
                    getAltura() / 2 + getPy());
        af.translate(getPx(), getPy());
```

```
        g.drawImage(getImagem().getImage(), af, null);

    }
...
```

Esse método de desenho parece bem melhor que a forma que fizemos para desenhar o tiro. Note que não inserimos nossas transformações no `Graphics2D`, apenas a passamos como parâmetro no método `drawImage`.

O método `divide`, além de achar o próximo tamanho do asteroide, retorna falso se ele não puder mais ser dividido.

```
public boolean divide() {
    switch (tamanho) {
    case GRANDE:
        tamanho = Tamanho.MEDIO;
        setImagem(Recursos.getImagem(
                    Recursos.Imagem.AST_B));
        break;
    case MEDIO:
        tamanho = Tamanho.PEQUENO;
        setImagem(Recursos.getImagem(
                    Recursos.Imagem.AST_C));
        break;
    case PEQUENO:
        tamanho = Tamanho.POEIRA;
    default:
        break;
    }

    return tamanho != Tamanho.POEIRA;
}
```

Além de mudarmos a imagem (e consequentemente o tamanho do elemento), o tamanho também influencia nos pontos que o jogador recebe:

```
...
    public short getPonto() {
        switch (tamanho) {
        case GRANDE:
```

```
            return 10;
        case MEDIO:
            return 15;
        case PEQUENO:
            return 20;
        case POEIRA:
        default:
            return 0;
        }
    }
}
```

O método `getPonto` garante que as pedras menores deem mais pontos que as maiores. Por último, mas não menos importante, nossa classe `Nave`.

Dessa vez, nossa nave se move pela tela, vagando pela galáxia sem freio. Então, quando o jogador acelera em uma direção, ele fica nessa direção até acelerar em outra, ou colidir com outro objeto.

```
public class Nave extends Elemento {

    public static final float ROTACAO_VEL = 5f;
    public static final float LIMITE_VEL = 9f;

    private float angulo;
    private float velEmX;
    private float velEmY;

    private short pontos;
    private short erros;
    private short seguidos;
```

Para controlar a velocidade da rotação da nave, usamos `ROTACAO_VEL` e, para ela não ganhar velocidade infinitamente, `LIMITE_VEL`. Para ela deslizar sobre a tela como um disco de hóquei, utilizamos `velEmX` e `velEmY`, que mantêm as velocidades de movimentação nos eixos. Além de somar os `pontos` do jogador, somamos os `erros` e os acertos `seguidos`.

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

```
    if (!isAtivo())
        return;
    setPx(getPx() + velEmX);
    setPy(getPy() + velEmY);
}
```

Mantemos a nave sempre em movimento atualizando sua posição. Em nossa terceira versão do método `desenha`, também usamos apenas um `AffineTransform` para guardamos as transformações anteriores. Entretanto, essa versão leva vantagem, porque não criamos um novo objeto, apenas usamos `afAnterior` como recipiente da referência do objeto já existente.

```
@Override
public void desenha(Graphics2D g) {
    if (!isAtivo())
        return;

    float rad = (float) Math.toRadians(getAngulo());

    afAnterior = g.getTransform();

    g.rotate(rad,
            getLargura() / 2 + getPx(),
            getAltura() / 2 + getPy());

    g.translate(getPx(), getPy());
    g.drawImage(getImagem().getImage(), 0, 0, null);

    g.setTransform(afAnterior);
}
```

Os pontos são multiplicados pela quantidade de tiros seguidos, que sempre volta para o quando o jogador erra ou é atingido. Consideremos que o jogador errou o tiro quando ele sair da tela, então incrementamos o contador de erros.

```
...
    public void somaPontos(short p) {
        seguidos++;
```

```
        pontos += p * seguidos;
    }

    public void errou() {
        seguidos = 0;
        erros++;
    }
```

Consertamos nossa nave, e agora ela pode se mover. Porém, sem *hyperdrive*, ela não pode alcançar a velocidade da luz, então limitamos a velocidade máxima nos métodos `setVelEmX` e `setVelEmY`.

```
...
    public float getVelEmX() {
        return velEmX;
    }

    public void setVelEmX(float vel) {
        if (vel > LIMITE_VEL)
            vel = LIMITE_VEL;
        else if (vel < -LIMITE_VEL)
            vel = -LIMITE_VEL;

        this.velEmX = vel;
    }

    public float getVelEmY() {
        return velEmY;
    }

    public void setVelEmY(float vel) {
        if (vel > LIMITE_VEL)
            vel = LIMITE_VEL;
        else if (vel < -LIMITE_VEL)
            vel = -LIMITE_VEL;

        this.velEmY = vel;
    }
```

```
    //... Outros getters e setters

    public void danos() {
        errou();
        velEmX = velEmX * -0.5f;
        velEmY = velEmY * -0.5f;
    }

}
```

Quando ocorrer colisões da nave com os asteroides, ela sofrerá danos e sua punição será na pontuação com um efeito de impacto fazendo-a perder velocidade. Para isso, chamamos o método `errou()`, e multiplicamos `velEmX` e `velEmY` por `-0.5`.

> **Nota**
>
> Asteroids guardava apenas três letras devido à limitação de hardware da época, mas não consigo imaginar o motivo de os diversos jogos de fliperama que vieram depois continuarem aceitando apenas três.

9.2 Dividir para destruir

Mesmo que não tenha som no espaço, nosso jogo terá explosões e, para não ficar feio perante o jogo original – no qual a nave do jogador se parte em pedaços pelo cosmos –, teremos duas animações básicas: uma para quando um tiro acerta um asteroide, e outra quando as rochas espaciais acertam nossas naves.

A primeira sequência de imagens (`explosao_asteroid.png`) é para quando um tiro atinge um asteroide, e a segunda (`colisao_asteroid.png`) é para quando a nave colide com um asteroide.

Fig. 9.2: Colisão da nave com asteroides

Fig. 9.3: Destruindo asteroides

Nossas explosões são controladas pela nova classe `Animacao.java`, que apenas percorre coluna por coluna do nosso sprite com um pequeno intervalo entre cada quadro.

```
public class Animacao extends Elemento {

    private short coluna;
    private short temporizador;

    public Animacao(ImageIcon imagem) {
        super.setImagem(imagem);
    }

    @Override
    public void atualiza() {
        if (!isAtivo())
            return;

        if (coluna == 6) {
            setAtivo(false);
        } else if (temporizador == 2) {
            coluna++;
            temporizador = 0;
        } else {
```

```
            temporizador++;
        }
    }
```

Utilizamos `coluna` para obter a imagem que queremos desenhar do sprite, e `temporizador` para controlar o tempo de sua exibição. Nossos dois sprites são simples, ambos com uma linha e seis colunas. Assim, quando nosso `temporizador` for igual a 2, incrementamos a coluna do sprite, e depois de exibirmos a última, encerramos a animação.

```
@Override
    public void desenha(Graphics2D g) {
        if (!isAtivo())
            return;

        // Largura da moldura
        int largMoldura = getImagem().getIconWidth() / 6;

        // Largura e altura do recorte da imagem
        int largImg = largMoldura * coluna;
        int altImg = getImagem().getIconHeight();

        g.drawImage(getImagem().getImage(),
                    getPx(), getPy(),
                    getPx() + largMoldura,
                       getPy() + altImg, largImg,
                    0, largImg + largMoldura,
                    altImg, null);
    }

    @Override
    public void setAtivo(boolean ativo) {
        super.setAtivo(ativo);
        coluna = 0;
        temporizador = 0;
    }
}
```

Para desenhar cada quadro, usamos nossa já conhecida fórmula para ob-

ter a parte a ser desenhada do sprite. Quando ativamos o elemento, reiniciamos a animação, assim reaproveitamos o mesmo objeto por um lado; por outro, interrompemos a animação caso ela ainda não tenha terminado.

Ao ser atingido, o asteroide se divide em dois até que não possa mais ser dividido. Em vez de criarmos novas instâncias da classe `Asteroide` a cada divisão durante o jogo, criamos um número equivalente já no início e as deixamos inativadas. Também reaproveitamos a instância do objeto ao invés de destruí-lo.

Como desenhamos apenas três tamanhos de asteroides (grande, médio e pequeno), para cada elemento precisaremos de mais 3, tendo um total de 4 novas instâncias ao total. Então, se iniciarmos o jogo com 5 rochas grandes, precisamos deixar espaço no array para 20 (5 * 4).

Fig. 9.4: Dividir e reaproveitar

Para cada `Asteroide` que teremos na tela, precisamos de quatro novas instâncias, que são representados na imagem pelas rochas com a frase "Novo".

Mesmo se nós não estivéssemos economizando memória, evitando criar novos objetos, nosso jogo, por ter tão poucos elementos, não ficaria lento. Essa medição é feita por meio da taxa de quadros por segundo, que pode oscilar, mas não ficar abaixo do que foi definido – nossos 20 quadros por segundo (FPS).

Como é bem provável que o jogo que você vai desenvolver fique tão grande que precise fazer essa medição, dedicamos a última parte da próxima seção para falar disso.

9.3 Jogando o código

Vimos os elementos que compõem nosso jogo, agora veremos como a classe `JogoCenario.java` orquestra essa sinfonia espacial. Vamos gerenciar nossos dois jogadores representados por `navJogUm` e `navJogDois` de forma individual. Se tivéssemos mais jogadores, faríamos diferente, utilizando um array.

Começamos o jogo com 10 asteroides grandes (`qtdeInicial`), mas a quantidade final de rochas na tela pode chegar a quatro vezes a quantidade inicial.

```
public class JogoCenario extends CenarioPadrao {

    ...
    private Nave navJogUm;
    private Nave navJogDois;

    private int qtdeInicial = 10;
    private int contadorTiro;
    private int contadorAsteroides;

    private Tiro[] tiros = new Tiro[16];
    private Asteroide[] aerolitos =
                new Asteroide[qtdeInicial * 4];

    private Animacao colisaoAst;
    private Animacao explosaoAst;

    ...
    private float graus;
    private boolean ampliar;
    private boolean reduzir = false; // Apenas exemplo
    private final float escala = 5 / 100f;
    // 5% do tamanho da tela
```

```
    private static final AffineTransform AF_VAZIO =
                        new AffineTransform();
```

Durante o jogo, usamos apenas dois objetos para gerenciar nossas duas animações, um para cada tipo de colisão. Então, embora elas tenham uma curtíssima duração, algumas vezes a animação é interrompida e começa em outra posição da tela. No modo de dois jogadores, quando as naves colidirem entre si, definimos `ampliar` como verdadeiro, aplicando um efeito de zoom de 5% na tela do jogo, valor da variável `escala`.

Deixamos um exemplo de como aplicar o efeito inverso e, para testá-lo, basta alterar `reduzir` para `true`. Utilizamos `AF_VAZIO` para desfazer o efeito de aproximação.

A principal diferença do método `carregar` deste jogo e do jogo anterior é que usamos imagens, todas obtidas por meio da classe `Recursos.java`, e nossas rochas são criadas com velocidade, ângulo e posição randômicas.

```
// método carregar:
fundo = Recursos.getImagem(Recursos.Imagem.FUNDO);

texto.setCor(Color.WHITE);

colisaoAst = new Animacao(
Recursos.getImagem(Recursos.Imagem.COLISAO_AST));
explosaoAst = new Animacao(
Recursos.getImagem(Recursos.Imagem.EXPLOSAO_AST));

for (int i = 0; i < tiros.length; i++) {
    tiros[i] = new Tiro();
    tiros[i].setVel(15);
    tiros[i].setImagem(Recursos.
                        getImagem(Recursos.Imagem.TIRO_A));
}

for (int i = 0; i < aerolitos.length; i++) {
    aerolitos[i] = new Asteroide();
    aerolitos[i].setImagem(Recursos.
                            getImagem(Recursos.Imagem.AST_A));
```

```
    if (i < qtdeInicial) {
        aerolitos[i].setAtivo(true);
        aerolitos[i].setVel(rand.nextInt(3) + 3);
        aerolitos[i].setAngulo(rand.nextInt(360));
        aerolitos[i].setPx(rand.nextInt(largura));
        aerolitos[i].setPy(rand.nextInt(altura));

        contadorAsteroides++;
    }
}
```

As naves dos jogadores iniciam no centro da tela, sendo que a nave do segundo jogador só ficará ativa se a opção dois jogadores – valor salvo em `Jogo.numeroJogadores` – for escolhida na tela inicial.

O segundo jogador pode entrar a qualquer momento durante o jogo pressionando o botão de tiro, configurado para o clique do botão principal do mouse.

```
navJogUm = new Nave();
navJogUm.setAtivo(true);
navJogUm.setImagem(Recursos.
                   getImagem(Recursos.Imagem.JOGADOR_A));

navJogDois = new Nave();
navJogDois.setAtivo(Jogo.numeroJogadores > 0);
navJogDois.setImagem(Recursos.
                        getImagem(Recursos.Imagem.JOGADOR_B));

Util.centraliza(navJogUm, largura, altura);
Util.centraliza(navJogDois, largura, altura);

navJogUm.setPy(navJogUm.getPy() - navJogUm.getAltura() / 2);
navJogDois.setPy(navJogDois.
                   getPy() + navJogDois.getAltura() / 2);
```

Após centralizarmos os jogadores, aplicamos um espaço entre eles para que não estejam colidindo no início do jogo. O método `atualizar` mudou bastante e, nessa mudança, cada jogador ganhou um método separado para

interações de controle, já que controlar pelo teclado difere do controle pelo mouse.

```
@Override
public void atualizar() {

    if (estado != Estado.JOGANDO) {
        return;
    }

    controlaJogadorUm();
    controlaJogadorDois();

    if (Util.colide(navJogUm, navJogDois)) {
        ampliar = true;

        navJogUm.setAngulo(
              MatUtil.corrigeGraus(navJogUm.getAngulo() + 90));
        navJogDois.setAngulo(
              MatUtil.corrigeGraus(navJogDois.getAngulo() - 90));

    } else {
        ampliar = false;
    }

    navJogUm.atualiza();
    navJogDois.atualiza();

    colisaoAst.atualiza();
    explosaoAst.atualiza();

    corrigePosicao(navJogUm);
    corrigePosicao(navJogDois);
```

Se detectarmos colisão entre as naves, além de ampliar a tela, bagunçamos suas navegações somando e subtraindo 90º graus de cada uma respectivamente. Para nossas naves não sumirem pelo universo ao saírem da tela, corrigimos sua posição fazendo-as aparecerem do lado oposto, isso graças ao método `corrigePosicao`, movido para classe `Util`.

Toda vez que um tiro colidir com um asteroide, posicionamos e reiniciamos a animação `explosaoAst`. Fazemos o mesmo com `colisaoAst` quando o asteroide colide com a nave.

```
for (int i = 0; i < contadorAsteroides; i++) {
    Asteroide ast = aerolitos[i];

    if (!ast.isAtivo())
        continue;

        for (Tiro tiro : tiros) {
            if (Util.colide(ast, tiro)) {
                tiro.setAtivo(false);

                explosaoAst.setAtivo(true);
                explosaoAst.setPx(ast.getPx());
                explosaoAst.setPy(ast.getPy());

                if (ast.divide())
                    novaParteAsteroide(ast);
                else
                    ast.setAtivo(false);

                tiro.getNave().somaPontos(ast.getPonto());
                break;
            }
        }
```

Em ambas as colisões, verificamos se é possível dividir o `Asteroide` antes de ativarmos mais um pedaço, mas a soma de pontos ocorre somente na colisão com o tiro. A colisão da nave com os asteroides é parecida, mas quando ela ocorre, a nave sofre danos.

Como o `Tiro` sabe de que nave foi disparado, é dele que obtemos a `Nave` para somar os pontos. A lógica está no próprio `Asteroide` que define o ponto de acordo com seu tamanho.

```
if (Util.colide(ast, navJogUm)) {
        navJogUm.danos();
```

```
            colisaoAst.setAtivo(true);
            colisaoAst.setPx(navJogUm.getPx());
            colisaoAst.setPy(navJogUm.getPy());

            if (ast.divide())
                novaParteAsteroide(ast);
            else {
                ast.setAtivo(false);
                continue;
            }

        } else if (Util.colide(ast, navJogDois)) {
            navJogDois.danos();
            colisaoAst.setAtivo(true);
            colisaoAst.setPx(navJogDois.getPx());
            colisaoAst.setPy(navJogDois.getPy());

            if (ast.divide())
                novaParteAsteroide(ast);
            else {
                ast.setAtivo(false);
                continue;
            }
        }

        ast.atualiza();
        corrigePosicao(ast);
    }
```

Se o asteroide já virou poeira, inativamo-lo e pulamos para a próxima iteração do laço. Apenas repetimos a condição trocando `navJogUm` por `navJogDois` para não separarmos essa parte do código em um novo método, facilitando a leitura.

No fim do laço, atualizamos e corrigimos a posição do nosso objeto. Entretanto, antes de terminarmos o método `atualizar`, precisamos verificar os tiros remanescentes.

```
for (Tiro tiro : tiros) {
            if (!tiro.isAtivo())
```

```
                continue;

            if (Util.saiu(tiro, largura, altura, 20)) {
                tiro.setAtivo(false);
                tiro.getNave().errou();
            } else
                tiro.atualiza();
        }

}
```

Verificamos se o tiro ainda está no limite da tela; se não estiver, consultamos a nave de origem do disparo para dar a triste notícia de que de ela errou. Note que essa é uma nova versão do método `Util.saiu`, que recebe um parâmetro a mais, usado somente para dar uma margem (de 20 pixels) do limite da tela, ficando assim:

```
public static boolean saiu(Elemento e, int largura,
                    int altura, int margem) {

    if (e.getPx() < -margem ||
            e.getPx() + e.getLargura() > largura + margem)
        return true;

    if (e.getPy() < -margem ||
            e.getPy() + e.getAltura() > altura + margem)
        return true;

    return false;
}
```

Hora de nos concentrarmos nos métodos menores, começando pela forma de pilotar cada nave.

```
public void controlaJogadorUm() {
    graus = navJogUm.getAngulo();

    if (Jogo.controleTecla[Jogo.Tecla.ESQUERDA.ordinal()])
        graus -= Nave.ROTACAO_VEL;
```

```
    else if (Jogo.controleTecla[Jogo.Tecla.DIREITA.ordinal()])
        graus += Nave.ROTACAO_VEL;

    navJogUm.setAngulo(graus);
```

A rotação ficou mais simplificada, já que passamos alguns métodos para a classe `MatUtil`. Quando o piloto/jogador pressionar para cima, a nave acelerará na direção em que estiver apontada.

Ele pode atirar ao mesmo tempo em que acelera, então, para não ficar estranho, é melhor que o tiro seja mais rápido que a nave.

```
if (Jogo.controleTecla[Jogo.Tecla.CIMA.ordinal()]) {
        float vx = navJogUm.getVelEmX();
        float vy = navJogUm.getVelEmY();

        vx += MatUtil.moveAnguloEmX(navJogUm.getAngulo());
        vy += MatUtil.moveAnguloEmY(navJogUm.getAngulo());

        navJogUm.setVelEmX(vx);
        navJogUm.setVelEmY(vy);

    }

    if (Jogo.controleTecla[Jogo.Tecla.BC.ordinal()]) {
        adicionarTiro(navJogUm);
        Jogo.liberaTecla(Jogo.Tecla.BC);
    }
}
```

Note que o método `adicionarTiro` agora recebe como parâmetro a `Nave` responsável pelo disparo. Trocando o teclado pelo mouse, o controle do segundo piloto ficou assim:

```
public void controlaJogadorDois() {

    if (!navJogDois.isAtivo()) {
        if (Jogo.controleTecla[Jogo.Tecla.MOUSE_A.ordinal()]) {
            navJogDois.setAtivo(true);
            Jogo.liberaTecla(Jogo.Tecla.MOUSE_A);
```

```
        }

        return;
    }

    graus = MatUtil.calculaDirecao(Jogo.pxyMouse.x,
                        Jogo.pxyMouse.y,
                        navJogDois.getPx(),
                        navJogDois.getPy());

    graus = MatUtil.corrigeGraus(graus);
    navJogDois.setAngulo(graus);
```

Se a nave não estiver ativa (modo de apenas um jogador), e o jogador pressionar o botão de disparo, a nave aparecerá no jogo; caso contrário, ignoramos o restante do código. Além da função `Math.atan2` ter ido parar dentro da `MatUtil.calculaDirecao`, a grande novidade inclui que usamos a posição do ponteiro do mouse na tela para rotacionar a nave. Então, a nave do segundo jogador aponta na direção do ponteiro do mouse.

```
if (Jogo.controleTecla[Jogo.Tecla.MOUSE_A.ordinal()]) {
        adicionarTiro(navJogDois);
        Jogo.liberaTecla(Jogo.Tecla.MOUSE_A);
    }

    if (Jogo.controleTecla[Jogo.Tecla.MOUSE_B.ordinal()]) {
        float vx = navJogDois.getVelEmX();
        float vy = navJogDois.getVelEmY();

        vx += MatUtil.moveAnguloEmX(navJogDois.getAngulo());
        vy += MatUtil.moveAnguloEmY(navJogDois.getAngulo());

        navJogDois.setVelEmX(vx);
        navJogDois.setVelEmY(vy);
    }
}
```

O clique do botão principal efetua disparos, e do botão secundário acelera a nave. Vale lembrar que, ao chamar `Jogo.liberaTecla`, impedimos

disparos seguidos mesmo se o jogador deixar pressionado o botão de disparo. Falando em disparos:

```
private void adicionarTiro(Nave jogador) {
    if (contadorTiro > 1)
        contadorTiro--;
    else
        contadorTiro = tiros.length - 1;

    Tiro t = tiros[contadorTiro];
    if (t.isAtivo())
        return;

    t.setNave(jogador);
    t.setAngulo(jogador.getAngulo());

    Util.centraliza(t, jogador);

    t.setAtivo(true);
}
```

Dica

Limitamos nossa jogabilidade aos cliques primários e secundários, mas a maioria dos mouses tem o botão de rolagem (e alguns ainda mais botões), que pode aumentar as ações do jogador.

Para uma versão mais sofisticada do jogo, onde cada nave teria seu próprio tiro ou simplesmente um controle individual deles, cada objeto teria seu próprio array de tiros. Porém, você pode aumentar o tamanho do array compartilhado para que as naves disparem à vontade.

Fazemos a divisão do asteroide quando ele colide ativando e configurando um `Asteroide` inativo e reconfigurando o objeto anterior. Isso ficou a cargo do método `novaParteAsteroide`, que só é chamado se ele ainda puder ser dividido.

```
private void novaParteAsteroide(Asteroide ast) {

    Asteroide novoAst = aerolitos[contadorAsteroides];

    novoAst.setAtivo(true);
    novoAst.setPx(ast.getPx());
    novoAst.setPy(ast.getPy());

    novoAst.setImagem(ast.getImagem());
    novoAst.setTamanho(ast.getTamanho());

    novoAst.setVel(rand.nextInt(3) + 5);

    float angulo = ast.getAngulo();

    ast.setAngulo(MatUtil.corrigeGraus(angulo + 90));
    novoAst.setAngulo(MatUtil.corrigeGraus(angulo - 90));

    contadorAsteroides++;
}
```

Pegamos o próximo elemento da fila (`novoAst`) e copiamos algumas informações do asteroide abatido (`ast`), que já está com o próximo `tamanho` (que pode ser `MEDIO` ou `PEQUENO`) definido. Por fim, aumentamos as chances do novo `ast` ter uma velocidade maior, e colocamos ambos em ângulos opostos.

O último método da classe `JogoCenario.java`, nosso método `desenhar`:

```
//método desenhar:
g.setTransform(AF_VAZIO); // Comentar para um zoom cada vez maior

if (ampliar) {
    g.scale(1 + escala, 1 + escala); // 1,05
    g.translate(-largura * escala / 2f + 1,
           -altura * escala / 2f + 1); // Centralizar

} else if (reduzir) {
    // Apenas exemplo
```

```
    g.setColor(Color.WHITE);
    g.fillRect(0, 0, largura, altura);
    g.scale(1 - escala, 1 - escala); // 1 - 0,05 = 0,95
    g.translate(largura * escala / 2f + 1,
           altura * escala / 2f + 1); // Centralizar
}
```

Primeiro zeramos qualquer transformação anterior usando `AF_VAZIO`, depois verificamos se vamos escalonar nossos desenhos em 5% do tamanho normal usando `scale` e, então, `translate` para centralizar a tela ampliada. Não usamos o efeito de redução, ficando apenas como exemplo.

Se não zerarmos as transformações anteriores, os efeitos se acumulam gerando um zoom cada vez maior. Ficou curioso? Basta comentar a linha.

```
g.drawImage(fundo.getImage(), 0, 0, null);

texto.desenha(g, "Tupã | " + navJogUm.getPontos(), 10, 20);
texto.desenha(g, "Îasy | " + navJogDois.getPontos(),
                largura - 120, 20);

if (navJogUm.getSeguidos() > 2)
    texto.desenha(g, "x" + navJogUm.getSeguidos(), 10, 40);

if (navJogDois.getSeguidos() > 2)
    texto.desenha(g, "x" + navJogDois.getSeguidos(),
                    largura - 120, 40);
```

Desenhamos a imagem de fundo e a pontuação dos jogadores. Estamos usando praticamente a mesma fonte, tamanho e cor, então, neste caso, não faz diferença chamar `g.drawString` ou `texto.desenha`.

A partir do terceiro acerto seguido, exibimos a quantidade de acertos seguidos que cada nave fez. Em seguida, desenhamos os tiros e aerólitos ativos.

```
for (int i = 0; i < tiros.length; i++) {
    if (tiros[i].isAtivo())
        tiros[i].desenha(g);
}
```

```
for (int i = 0; i < contadorAsteroides; i++) {
    if (aerolitos[i].isAtivo())
        aerolitos[i].desenha(g);
}

navJogUm.desenha(g);
navJogDois.desenha(g);

colisaoAst.desenha(g);
explosaoAst.desenha(g);
```

Para ficar sobre os outros elementos, desenhamos por último as naves dos jogadores e as animações de colisão e explosão.

Fig. 9.5: Exibindo acertos seguidos

Dica

Você pode deixar o placar com mais cara de árcade usando: `String.format("Tupã | %04d", navJogUm.getPontos())`, em que `%04d` representará a pontuação com `4` dígitos, preenchendo os espaços vazios com `0`.

Ao longo dos capítulos, não fizemos nenhuma grande alteração na classe `Jogo.java`. Mudamos o tamanho da tela para alguns jogos, adicionamos botões ou novas entradas em outros, mas desta vez fizemos muitas alterações na nossa classe `Jogo.java`. Começamos pelo controle do segundo jogador.

```
...
public enum Tecla {
    CIMA, BAIXO, ESQUERDA, DIREITA, BA, BB, BC,
                    MOUSE_A, MOUSE_B
}

...
public static int numeroJogadores;
public static final Point pxyMouse = new Point();
```

Para controlar o segundo jogador, que estará disponível se `numeroJogadores` for igual a 1 (segunda opção do menu), adicionamos mais dois valores ao nosso `enum Tecla`: `MOUSE_A` e `MOUSE_B`. Também, usamos a variável `pxyMouse` para guardar a posição do mouse na tela, usando um `java.awt.Point`.

Em nosso construtor, adicionamos ouvintes para detectar não só o clique (`MouseListener`), como também o movimento do mouse (`MouseMotionListener`).

```
public Jogo() {
...
    tela.addMouseMotionListener(new MouseMotionListener() {

        @Override
        public void mouseMoved(MouseEvent e) {
            pxyMouse.x = e.getX();
            pxyMouse.y = e.getY();
        }

        @Override
        public void mouseDragged(MouseEvent e) {
        }
    });
```

Observe que em nossa implementação do `mouseMoved`, poderíamos obter um objeto `Point` diretamente com `e.getPoint()`, porém, em vez disso, armazenamos os valores inteiros diretamente para reaproveitarmos o mesmo objeto.

De forma similar ao que fazemos para controlar as teclas pressionadas, usamos `mouseReleased` e `mousePressed` para verificar os cliques do mouse. Se for o botão principal (`MouseEvent.BUTTON1`), a nave atira; se for qualquer outro, ela se move.

```
tela.addMouseListener(new MouseListener() {

        @Override
        public void mouseReleased(MouseEvent e) {
            if (e.getButton() == MouseEvent.BUTTON1)
                controleTecla[Tecla.MOUSE_A.ordinal()] = false;
            else
                controleTecla[Tecla.MOUSE_B.ordinal()] = false;
        }

        @Override
        public void mousePressed(MouseEvent e) {
            if (e.getButton() == MouseEvent.BUTTON1)
                controleTecla[Tecla.MOUSE_A.ordinal()] = true;
            else
                controleTecla[Tecla.MOUSE_B.ordinal()] = true;
        }

        @Override
        public void mouseExited(MouseEvent e) {
        }

        @Override
        public void mouseEntered(MouseEvent e) {
        }

        @Override
        public void mouseClicked(MouseEvent e) {
        }
    });
...
}
```

Ignoramos outras ações do mouse, que podem ser relembradas nos capí-

tulos 1 e 3. Embora este seja nosso único jogo multijogador, a lógica aplicada aqui pode ser usada em quase todos os jogos que já fizemos.

Deixamos para o final uma melhoria opcional, que usamos para testar a taxa de quadros do nosso jogo.

Fizemos nosso jogo prevendo que teríamos 20 frames por segundo (`private static final int FPS = 1000 / 20`), mas será que realmente temos em um segundo 20 execuções dos métodos `atualizar` e `desenhar`?

Para tirar essa dúvida, melhoramos nosso motor de jogo, e agora conseguimos ver quantos quadros são desenhados por segundo.

```
...
public static int atraso = 0;
public static int somaFPS = 0;

private static final SimpleDateFormat sdf =
                new SimpleDateFormat("ss:SSSS");

private static final boolean depurar = true;
```

Na variável `atraso`, armazenamos os milésimos de segundo que se passaram entre o início e o fim do processamento do jogo e, na variável `somaFPS`, quantos processamentos temos a cada 1 segundo ou 1000 milésimos de segundo. Para ainda mais informação, usamos o `SimpleDateFormat` para exibir o tempo corrente (obtido com `System.currentTimeMillis()`) em segundos e milissegundos, quando `depurar` for verdadeiro.

Não importa o quanto seu processador seja rápido, ele vai precisar de alguns milissegundos (nanossegundos se este livro demorou muito para ser escrito) para processar (atualizar e desenhar) nosso jogo. Utilizamos `agora` para armazenar o tempo no início do processo, e depois medirmos os milésimos de segundo que se passaram.

```
public void iniciarJogo() {
    long agora;
    int contadorFPS = 0;
    long prxAtualizacao = 0;
```

```
    long prxSegundo = System.currentTimeMillis() + 1000;

    while (true) {
        agora = System.currentTimeMillis();

        if (agora >= prxAtualizacao) {

            verificaTeclas();

            g2d.setColor(Color.BLACK);
            g2d.fillRect(0, 0, JANELA_LARGURA, JANELA_ALTURA);

            if (!Jogo.pausado)
                cenario.atualizar();

            cenario.desenhar(g2d);
```

Incrementamos `contadorFPS` a cada final de processamento, sendo que `somaFPS` recebe o valor de `contadorFPS` a cada 1 segundo, antes de ele ser zerado. A variável `prxAtualizacao` é nossa velha conhecida, usada para sabermos quando processar novamente nosso jogo.

Além de imprimirmos alguns dados no console, desenhamos no canto esquerdo inferior da tela o valor armazenado na variável `somaFPS`.

```
if (!Jogo.pausado)
        cenario.atualizar();

    cenario.desenhar(g2d);

    if (Jogo.pausado) {
        g2d.setColor(Color.WHITE);
        g2d.drawString("Pausado",
                        tela.getWidth() / 2 - 30, 30);
    }

    if (depurar) {
        g2d.setColor(Color.WHITE);
        g2d.drawString("FPS " + somaFPS, 10,
```

```
                        JANELA_ALTURA - 10);
    }

    tela.repaint();

    contadorFPS++;

    atraso = (int) (System.currentTimeMillis() - agora);
    prxAtualizacao = System.currentTimeMillis() + FPS;
}
```

Medimos o atraso na execução do jogo subtraindo, ao final do processamento, o tempo corrente pelo valor armazenado antes de iniciarmos o processamento. Assim, definimos o valor da variável `atraso`.

Toda vez que processamos o jogo (quando `agora` for maior ou igual a `prxAtualizacao`), incrementamos `contadorFPS` e, a cada segundo (quando `agora` for maior ou igual `prxSegundo`), armazenamos o resultado em `somaFPS` e reiniciamos a contagem.

```
if (depurar && agora >= prxSegundo) {
            System.out.println("Atraso = " + atraso);
            System.out.println("FPS = " + contadorFPS);
            System.out.println("Segundos: " + sdf.format(agora));

            somaFPS = contadorFPS;
            contadorFPS = 0;
            prxSegundo = System.currentTimeMillis() + 1000;
        }
    }
}
```

Dessa forma, quando `depurar` for verdadeiro, além de mostramos na tela a taxa de quadros por segundo, imprimirmos quantos milésimos de segundo custa para processar nosso jogo.

Se a contagem de quadros estiver menor que o esperado, faça testes aumentando seu `FPS`, de 20 para 21, por exemplo. Mas observe que a velocidade dos nossos jogos é baseada em pixels por FPS, então, a menos que esteja

aplicando um efeito de aceleração (ou desaceleração), mudar drasticamente a taxa de quadros fará o jogo parecer quebrado.

> **Dica**
>
> Embora não abordemos, outra forma comum de controlar a velocidade do jogo, independente do FPS, envolve levar em consideração o tempo antes e depois do processamento, e usar essa diferença (conhecida como *delta time*) na movimentação dos elementos.

Por ser um jogo pequeno, o atraso entre um processamento e outro é quase insignificante, mas saiba que existem diversas técnicas para compensar atrasos (curtos ou longos), que podem ser bem simples ou bem complexas.

Falando simplificadamente de duas dessas técnicas, a primeira é forçar mais o processador, fazendo o próximo processamento acontecer mais cedo, isso é, subtraindo o `atraso` do tempo de espera para a próxima atualização. Um exemplo seria `prxAtualizacao = System.currentTimeMillis() + FPS - atraso`.

A segunda técnica consiste em fazer o contrário da primeira: pular a fase de desenho – que geralmente consome mais recursos – do próximo processamento quando o jogo acumular muitos atrasos para aliviar o processador e tentar recuperar o tempo perdido. Mas isso somente quando o jogo não tem um gráfico extremamente simples com uma inteligência artificial complexa, ou um modo de múltiplos jogadores em rede.

Existe muita discussão sobre este assunto na internet e muitos tutoriais, como em http://www.pontov.com.br. Assim, recomendo procurar mais sobre, e você notará que a falta de consistência na taxa de quadros é um problema até mesmo para jogos recentes de grandes produtoras.

9.4 Codificando o jogo

Sem repetirmos o que foi visto no capítulo anterior, temos três novas classes: `MatUtil.java` (que está no pacote *base*), `Recursos.java` e

`Animacao.java`. Criamos `MatUtil.java` para simplificar algumas chamadas aos métodos da classe `Math`, e continuamos utilizando somente graus para controlar nossos ângulos.

Nossa classe `Recursos.java` concentra todas as imagens do jogo, que estão na pasta `imagens` na raiz do projeto, mas também pode ser usada para concentrar outros recursos do jogo, como sons e, quem sabe, vídeos.

Para controlar nossa pequena animação de explosão, usada nas colisões, criamos a classe `Animacao.java`, que percorre um sprite simples dentro de um intervalo pré-determinado.

Agora, nosso `Asteroide.java` vaga sem rumo pelo espaço e, ao colidir com a nave ou ser acertado por um tiro, se divide indo do tamanho grande para o médio, do médio para o pequeno, e depois virando poeira.

O `Tiro.java` agora sabe de que nave foi disparado, influenciando na pontuação do jogador. Quando algum jogador alcançar mais de dois acertos seguidos, apresentamos uma indicação visual de que ele está indo bem.

Nossa `Nave.java`, que começa parada no centro da tela, agora se move por ela e, uma vez que tenha saído do lugar, requer habilidade para executar manobras e perdem o controle ao colidirem. Cada nave recebeu um nome que pegamos emprestado da cultura Tupi Guarani.

O jogo ganhou um modo para dois jogadores: um controlado pelo teclado e outro pelo mouse. Também aplicamos um efeito de aproximação na tela como um todo, mudando o desenho final. Essas são as novidades na classe `JogoCenario.java`.

Depois de tudo isso, alteramos a classe `Jogo.java` para termos mais informações acerca da constância do jogo, então melhoramos nosso motor e agora conseguimos ver quantos quadros são desenhamos por segundo.

Adicionamos o método `corrigePosicao` e uma nova versão do método `saiu` na classe `Util.java`. Também temos uma classe `RotacaoTeste.java`, que faz a mesma lógica da classe do capítulo anterior com imagens.

Confira tudo isso em https://github.com/logicadojogo/fontes/tree/master/Cap09.

Não fizemos aqui

- Modo jogador versus jogador;
- Tiro diferente para o segundo jogador.

Melhore você mesmo

- Exibir placar final na introdução do jogo;
- Contar máximo de acertos seguidos.

9.5 Resumo

Se nossa versão chega aos pés do jogo original, só você pode dizer, mas com certeza honramos o nome Asteroids adicionando conceitos de física, um pequeno efeito de animação e a possibilidade de dois jogadores.

Temos pequenos problemas, como por exemplo, o jogador com o mouse leva vantagem já que consegue girar a nave muito mais rápido que o jogador no teclado. Além de não termos dado invulnerabilidade momentânea aos jogadores no início do jogo e ao serem atingidos, causando múltiplas colisões.

Já vimos formas de implementar essas funcionalidades em outros jogos e, ao juntar e misturar os jogos de cada capítulo, acabará criando novos jogos.

Capítulo 10

Última fase, último chefe e próximos jogos

Vimos bastante coisa até aqui, foram 6 jogos (sem contarmos os protótipos), mais de 44 classes, além de algumas músicas e imagens. Você tem código suficiente para recriar outros jogos de *Arcade*, muitos deles apenas misturando ou alterando os jogos de um ou mais capítulos. Além, é claro, de código para criar seus próprios jogos originais, inclusive em outras plataformas e linguagens de programação.

Se você ouviu nossas súplicas e personalizou, melhorou ou ampliou algum jogo visto neste livro, por favor, contribua submetendo suas alterações para o repositório que criamos exatamente para que você fizesse isso, em: https://github.com/logicadojogo. Toda e qualquer contribuição será benquista.

10.1 Desafios

Depois de construirmos nossa versão do jogo, no final de cada capítulo, deixamos alguns desafios para você leitor. A lógica para resolvê-los foi mostrada em capítulos posteriores, e você pode ter obtido o mesmo resultado usando um código ou lógica diferente. Mesmo assim, vamos rever, de forma resumida, alguns pontos que merecem um pouco mais de destaque.

No capítulo 2, não criamos as barreiras parecidas com a do jogo original. Podemos fazer isso da mesma forma que criamos cenários no capítulo 4, então, na classe `Nivel.java` criamos o seguinte array:

```
public class Nivel {
    //11 por 8, 12 por 8 e 8 por 8
    public static char[][][] niveis = {
            {
                    { ' ', ' ', '0', ' ', ' ', ' ', ' ', ' ', '0', ' ', ' '},
                    { ' ', ' ', ' ', '0', ' ', ' ', ' ', '0', ' ', ' ', ' '},
                    { ' ', ' ', '0', '0', '0', '0', '0', '0', '0', ' ', ' '},
                    { ' ', '0', '0', ' ', '0', '0', '0', ' ', '0', '0', ' '},
                    { '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0'},
                    { '0', ' ', '0', '0', '0', '0', '0', '0', '0', ' ', '0'},
                    { '0', ' ', '0', ' ', ' ', ' ', ' ', ' ', '0', ' ', '0'},
                    { ' ', ' ', ' ', '0', '0', ' ', '0', '0', ' ', ' ', ' '} },

            {
                    { ' ', ' ', ' ', ' ', '0', '0', '0', '0', ' ', ' ', ' ', ' '},
                    { ' ', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', ' '},
                    { '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0'},
                    { '0', '0', '0', ' ', ' ', '0', '0', ' ', ' ', '0', '0', '0'},
                    { '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0'},
                    { ' ', ' ', ' ', ' ', '0', '0', ' ', ' ', '0', '0', ' ', ' '},
                    { ' ', ' ', ' ', '0', '0', ' ', '0', '0', ' ', ' ', ' ', ' '},
                    { '0', '0', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '0', '0'} },

            {
                    { ' ', ' ', ' ', '0', '0', ' ', ' ', ' '},
                    { ' ', ' ', '0', '0', '0', '0', ' ', ' '},
                    { ' ', '0', '0', '0', '0', '0', '0', ' '},
                    { '0', '0', ' ', '0', '0', ' ', '0', '0'},
                    { '0', '0', '0', '0', '0', '0', '0', '0'},
                    { ' ', '0', ' ', '0', '0', ' ', '0', ' '},
                    { '0', ' ', ' ', ' ', ' ', ' ', ' ', '0'},
                    { ' ', '0', '0', ' ', ' ', '0', '0', ' '} } };
}
```

Fig. 10.1: Barreiras do jogo

As barreiras poderiam ser mais simples, mas optamos por homenagear o jogo original. Modificamos o método `carregarJogo` da classe

`Jogo.java`, lembrando de que neste capítulo ainda não tínhamos a classe `JogoCenario.java`:

```
private void carregarJogo() {

    int total = 0;
    int _LARG = 10;

    for (int i = 0; i < Nivel.niveis.length; i++) {
        char[][] n = Nivel.niveis[i];

        for (int linha = 0; linha < n.length; linha++) {
            for (int coluna = 0; coluna < n[0].length; coluna++){
                if (n[linha][coluna] == '0')
                    total++;
            }
        }
    }

    nivel = new Elemento[total];

    for (int i = 0; i < Nivel.niveis.length; i++) {
        char[][] n = Nivel.niveis[i];

        for (int linha = 0; linha < n.length; linha++) {
            for (int coluna = 0; coluna < n[0].length; coluna++){
                if (n[linha][coluna] != ' ') {

                    Elemento e = new Elemento();
                    e.setAtivo(true);
                    e.setCor(Color.LIGHT_GRAY);

                    e.setPx(_LARG * coluna + 30 + (200 * i));
                    e.setPy(_LARG * linha +
                                    JANELA_ALTURA - 300);

                    e.setAltura(_LARG);
                    e.setLargura(_LARG);
```

```
                    nivel[--total] = e;
                }
            }
        }
    }
```

Cada pedaço da barreira será um objeto `Elemento`. Depois, só precisamos desenhar e verificar a colisão dos `tiros` com a barreira:

```
for (Elemento e : nivel) {
    if (!e.isAtivo())
        continue;

    for (int i = 0; i < tiros.length; i++) {
        if (tiros[i].isAtivo() && Util.colide(tiros[i], e)) {
            e.setAtivo(false);
            tiros[i].setAtivo(false);
        }
    }

    g2d.setColor(e.getCor());
    e.desenha(g2d);
}
```

Gerando o seguinte resultado:

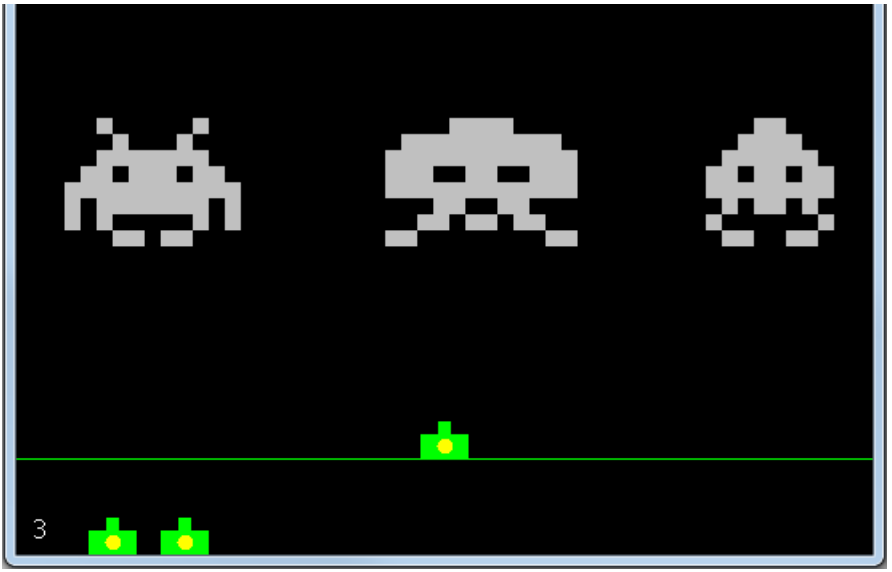

Fig. 10.2: Barreiras em homenagem ao jogo original

Para impedir o jogador de sair da tela, temos o método `saiu` da classe `Util.java` e, para encerrar o jogo quando o tanque perder todas as vidas, podemos usar o mesmo exemplo do capítulo 4 ou 5.

No capítulo 3, deixamos de calcular o ângulo da bola na rebatida, mas vimos como trabalhar com ângulo no capítulo 9. Para exibir as últimas pontuações no cenário de introdução do jogo, você pode usar nossa classe `Texto.java`, chamando o método `texto.desenha`, ou diretamente `g.drawString` (veja um exemplo mais adiante neste capítulo). Ficou por sua conta criar um modo de jogo seu.

No capítulo 4, para ir para o próximo nível quando o jogador ganhar, temos um bom exemplo no capítulo 5 e, para definir a posição inicial da serpente de acordo com a fase, fizemos algo parecido no capítulo 6. Inclusive, adicionando mais valores ao array de níveis para criar cenários com mais cores. Para criar seu próprio nível, você pode editar ou adicionar mais arrays bidimensionais ao array `niveis` da classe `Nivel.java`.

No capítulo 5, podemos adicionar um novo menu para ter a opção de jogar com e sem som, ou somente efeitos sonoros. Veja como exemplo o menu

do capítulo 4. Para manipular a música de fundo durante o jogo, você pode aumentar a batida por minuto de acordo com o nível do jogo, por exemplo:

```
seqSomDeFundo.setTempoInBPM(seqSomDeFundo.getTempoInBPM()
                            * nivel);
```

Para adicionar um botão para girar a peça no sentido horário, você pode mapear a tecla `espaço` para chamar o método `girarReposicionarPeca(true)` ou `girarPeca(true)`, lembrando que, no anti-horário, você passa o valor `false`. No próprio capítulo, deixamos dicas para achar ou criar sua própria música de fundo e efeitos sonoros.

No capítulo 6, tanto para ter a família de quatro fantasmas completa, reiniciar o cenário ao comer todas as pastilhas e criar uma cópia do cenário para não perder a configuração inicial, basta ver a versão mais avançada do jogo no capítulo 7. Para adicionar frutas que aumentem a pontuação do jogador, primeiro adicionamos um novo valor ao array `cenario`, responsável por montar o cenário do jogo, na classe `Nivel.java`. Para controlar a posição na grade em que a fruta vai aparecer, criamos um novo valor:

```
...
/** Fruta */
public static final int FT = 10;
...
```

Nosso valor será representado por `FT` e adicionado ao lado da posição inicial do jogador (representado por `PI`) no cenário. Depois criamos um novo elemento, que, além de representar a fruta no jogo, leva em conta o espaçamento superior que usamos para exibir os pontos do jogador:

```
private Elemento fruta = new Elemento() {
    @Override
    public void desenha(Graphics2D g) {
       if(!isAtivo())
           return;

       g.setColor(Color.RED);
       g.fillOval(getPx() - 4, getPy() + JogoCenario.ESPACO_TOPO,
```

```
                            getLargura() / 2,
                            getAltura() / 2);

        g.fillOval(getPx() + 4, getPy() + JogoCenario.ESPACO_TOPO,
                            getLargura() / 2,
                            getAltura() / 2);

        g.fillOval(getPx(), getPy() + 5 + JogoCenario.ESPACO_TOPO,
                            getLargura() / 2,
                            getAltura() / 2);
    }
};
```

De forma simples, desenhamos três bolas vermelhas para simular uma cereja. Então, configuramos o objeto `fruta` dentro do método `carregar`, ao percorrer o array `grade`:

```
...
} else if (grade[lin][col] == Nivel.FT) {
    fruta.setLargura(largEl);
    fruta.setAltura(largEl);
    fruta.setPx(converteInidicePosicao(col));
    fruta.setPy(converteInidicePosicao(lin));
}
```

No método `atualizar`, para ativar a fruta fazendo-a ficar visível, podemos usar um contador ou aleatoriamente desta forma:

```
...
if (!fruta.isAtivo() && rand.nextInt(1000) == 5) {
    fruta.setAtivo(true);
}

if (Util.colide(pizza, fruta)) {
    fruta.setAtivo(false);
    pontos += 100;
}
...
```

Se a fruta estiver ativa, o método `Util.colide` retornará verdadeiro. Logo, desativamos a fruta e damos 100 pontos para o jogador. Só não esqueça de desenhá-la antes dos demais elementos:

```
...
texto.desenha(g, "Pontos: " + pontos, 10, 20);
fruta.desenha(g);
pizza.desenha(g);
...
```

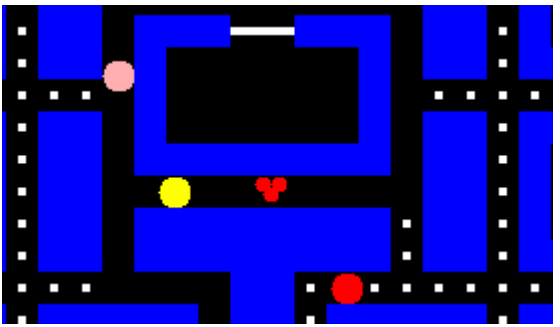

Fig. 10.3: Algo que parece uma cereja

No capítulo 7, trabalhamos com sprites e tiles, mas, para aplicar rotação nas imagens, olhe os exemplos no capítulo 9. Para apresentar os personagens na introdução do jogo, modificamos somente a classe `InicioCenario.java`:

```
...
private Pizza pizza;
private Legume[] inimigos;
private Elemento superPastilha;

@Override
public void carregar() {
...
    pizza = new Pizza();
    pizza.setVel(3);
```

```
    pizza.setAtivo(true);
    pizza.setDx(-1);
    pizza.setPx(largura + pizza.getLargura());
    pizza.setPy(200);

    superPastilha = new Elemento() {
        @Override
        public void desenha(Graphics2D g) {
            if (!isAtivo())
                return;

            g.drawImage(getImagem().getImage(),
                        getPx(),
                        getPy() + JogoCenario.ESPACO_TOPO, null);
        }
    };
```

Adicionamos o personagem principal, os inimigos e uma superpastilha, que é um `Elemento` comum; apenas modificamos o método `desenha` para levar em consideração o espaçamento no topo, também utilizado para desenhar os outros elementos. O objetivo é mostrar o personagem principal correndo dos inimigos da direita para a esquerda, até chegar à pastilha onde entrará no modo super, invertendo os papéis e fazendo os inimigos fugirem.

```
ImageIcon pepperoni = new ImageIcon("imagens/pepperoni.png");
    superPastilha.setAtivo(true);
    superPastilha.setPx(5);
    superPastilha.setPy(pizza.getPy());
    superPastilha.setImagem(pepperoni);
    superPastilha.setLargura(pepperoni.getIconWidth());
    superPastilha.setAltura(pepperoni.getIconHeight());

    inimigos = new Legume[Legume.Tipo.values().length];
    for (int i = 0; i < inimigos.length; i++) {
        inimigos[i] = new Legume(Legume.Tipo.values()[i]);
        inimigos[i].setVel(3);
        inimigos[i].setAtivo(true);
        inimigos[i].setDx(-1);
```

```
        inimigos[i].setPy(pizza.getPy());
        inimigos[i].setPx(largura +
                            (pizza.getLargura() * 2) * (i + 2));
    }
}
```

Todos os elementos ficarão no mesmo alinhamento horizontal do elemento `pizza` e, além de utilizarmos a imagem do `pepperoni.png` para a pastilha, atualizamos seu tamanho para que a detecção de colisão funcione corretamente. Configuramos todos os inimigos de uma só vez, adicionando espaçamento suficiente entre eles para aparecerem fora da tela, depois do personagem principal.

```
@Override
public void atualizar() {
...

    if (pizza.getPx() > largura * 2) {
        return;
    }

    pizza.atualiza();
    for (Legume legume : inimigos) {
        legume.atualiza();

        if (Util.colide(pizza, legume)) {
            legume.setModo(Legume.Modo.FANTASMA);
        }
    }
```

Os personagens só serão atualizados dentro de um limite que é o dobro da largura da tela para garantir que todos saiam de cena e não acabem congelados no meio do caminho. Se o objeto `pizza` colidir com um `legume`, o inimigo vira um fantasma, mas isso só acontecerá depois que o personagem colidir com a superpastilha, quando invertemos a direção deles.

```
if (Util.colide(pizza, superPastilha)) {
        pizza.setDx(1);
        pizza.setVel(pizza.getVel() + 2);
```

```
            superPastilha.setAtivo(false);
            for (Legume legume : inimigos) {
                legume.setDx(1);
                legume.setModo(Legume.Modo.FUGINDO);
            }
        }
    }

    @Override
    public void desenhar(Graphics2D g) {
        superPastilha.desenha(g);
        pizza.desenha(g);

        for (Legume legume : inimigos) {
            legume.desenha(g);
        }

        menuJogo.desenha(g);
    }
```

Por fim, antes de desenharmos o menu, desenhamos os novos elementos. Vimos como aplicar efeitos sonoros no capítulo 5, e deixamos por sua conta usar suas próprias imagens para personalizar o jogo.

No capítulo 8, se deixamos de movimentar a nave pela tela e controlar a nave com o mouse, foi para fazermos isso no capítulo 9, que também mostra como contar acertos seguidos para aumentar a pontuação do jogador e utilizar imagens.

Para criarmos formas de tiro que usem ângulos diferentes, utilizamos um `enum` para controlar o tipo de disparo:

```
enum TipoTiro {
    COMUM, DUPLO, TRIPLO
}

private void proximoTiro() {
    if (contadorTiro > 1)
        contadorTiro--;
    else
```

```
        contadorTiro = tiros.length - 1;
}
```

Como podemos ter mais de um disparo por vez, separamos do método `adicionarTiro` o contador de tiros, para poder chamá-lo de acordo com a quantidade de projéteis necessária.

```
private void adicionarTiro(float angulo, TipoTiro tipo) {

    if (TipoTiro.COMUM == tipo || TipoTiro.TRIPLO == tipo) {

        proximoTiro();

        Tiro t = tiros[contadorTiro];

        t.setAngulo(angulo);
        t.setPx(nave.getPx() +
                nave.getLargura() / 2 - t.getLargura() / 2);

        t.setPy(nave.getPy() +
                nave.getAltura() / 2 - t.getAltura() / 2);

        t.setAtivo(true);
    }
```

O tipo de tiro comum continua sendo um disparo central, o duplo também começa no centro, mas segue em direção oposta um do outro, e o triplo acaba sendo a mistura desses dois.

```
if (TipoTiro.DUPLO == tipo || TipoTiro.TRIPLO == tipo) {
        proximoTiro();
        Tiro tiroA = tiros[contadorTiro];

        proximoTiro();
        Tiro tiroB = tiros[contadorTiro];

        tiroA.setAngulo(angulo - 10);
        tiroA.setPx(nave.getPx() + nave.getLargura() / 2 -
                        tiroA.getLargura() / 2);
```

```
        tiroA.setPy(nave.getPy() + nave.getAltura() / 2 -
                        tiroA.getAltura() / 2);

        tiroB.setAngulo(angulo + 10);
        tiroB.setPx(nave.getPx() + nave.getLargura() / 2 -
                        tiroB.getLargura() / 2);

        tiroB.setPy(nave.getPy() + nave.getAltura() / 2 -
                        tiroB.getAltura() / 2);

        tiroA.setAtivo(true);
        tiroB.setAtivo(true);
    }
}
```

Estamos centralizando os tiros usando individualmente a largura e altura de cada um, isso só fará diferença se os objetos `Tiro` tiverem tamanhos diferentes. O `tiroA` recebe o ângulo da nave menos 10, já o `tiroB`, o ângulo da nave mais 10.

Fig. 10.4: Tiro comum, duplo e triplo

A mudança entre os tipos de tiro pode acontecer de diversas maneiras, como por exemplo, o jogador acertar uma quantidade de inimigos, ou algum item que ele possa conseguir acertar ou colidir, o famoso *power up*.

No capítulo 9, para termos um tiro diferente para o segundo jogador, além de mudanças nas propriedades do tiro como velocidade ou ângulo, podemos utilizar uma nova imagem. Para isso, já deixamos algumas coisas prontas. Começamos alterando o método `carregarImagens`, da classe `Recursos`, para que carregue um novo arquivo:

```
public static void carregarImagens() {
...
```

```
    imagens[Imagem.TIRO_A.ordinal()] =
                    new ImageIcon(DIR_IMG + "tiro.png");
    imagens[Imagem.TIRO_B.ordinal()] =
                    new ImageIcon(DIR_IMG + "tiro_b.png");
...
}
```

Depois, no método `adicionarTiro` na classe `JogoCenario`, definimos a imagem do tiro de acordo com o jogador:

```
private void adicionarTiro(Nave jogador) {
...
    if (navJogUm == jogador)
        t.setImagem(Recursos.getImagem(Recursos.Imagem.TIRO_A));
    else
        t.setImagem(Recursos.getImagem(Recursos.Imagem.TIRO_B));

    t.setAtivo(true);
}
```

Para o modo jogador versus jogador, dentro do método `atualizar` (ainda dentro da classe `JogoCenario`), caso não tenha saído da tela, atualizamos e verificamos se o `tiro` do jogador atingiu a nave oposta:

```
for (Tiro tiro : tiros) {
    if (!tiro.isAtivo())
        continue;

    if (Util.saiu(tiro, largura, altura, 20)) {
        tiro.setAtivo(false);
        tiro.getNave().errou();
    } else {
        tiro.atualiza();

        if (navJogUm != tiro.getNave()
            && Util.colide(tiro, navJogUm)) {

            tiro.setAtivo(false);
            navJogUm.danos();
```

```
            navJogUm.setAngulo(
                    MatUtil.corrigeGraus(
                        navJogUm.getAngulo() + 90));

        } else if (navJogDois != tiro.getNave()
                    && Util.colide(tiro, navJogDois)) {

            tiro.setAtivo(false);
            navJogDois.danos();
            navJogDois.setAngulo(
                    MatUtil.corrigeGraus(
                        navJogDois.getAngulo() - 90));
        }
    }
}
```

Para contar o máximo de acertos seguidos, faça algo parecido com o que fizemos com o contador de erros e, para exibir placar final na introdução do jogo, criamos um array estático para armazenar a pontuação máxima de cada jogador, algo parecido com `public static short[] jogadorPontos = new short[2]`. Depois de verificarmos a colisão entre asteroides e tiros, e somarmos os pontos dos jogadores, atualizamos a pontuação caso ela seja maior que a anterior.

```
for (Tiro tiro : tiros) {
    if (Util.colide(ast, tiro)) {
...
        tiro.getNave().somaPontos(ast.getPonto());

        short pontos = tiro.getNave().getPontos();

        if (navJogUm == tiro.getNave() &&
                    pontos > Jogo.jogadorPontos[0])
            Jogo.jogadorPontos[0] = pontos;
        else if (navJogDois == tiro.getNave() &&
                    pontos > Jogo.jogadorPontos[1])
            Jogo.jogadorPontos[1] = tiro.getNave().getPontos();
...
}
```

Modificamos a classe `InicioCenario.java` para exibir o placar com um efeito de rolamento vertical, utilizando a variável `novaPy` e um objeto `texto`.

```
private int novaPy;
private Texto texto = new Texto();

@Override
public void atualizar() {

    novaPy++;

    if (novaPy + 100 > altura)
        novaPy = -100;

...
}
```

Então, desenhamos a pontuação usando `novaPy` para definir o posicionamento no eixo Y.

```
@Override
public void desenhar(Graphics2D g) {
...
    g.setColor(Color.WHITE);
    texto.desenha(g, String.format("Tupã | %04d",
            Jogo.jogadorPontos[0]), largura / 4, novaPy);

    texto.desenha(g, String.format("Îasy | %04d",
             Jogo.jogadorPontos[1]), largura / 2 + 100, novaPy);

    menuJogo.desenha(g);
}
```

Obtendo o seguinte resultado:

Fig. 10.5: O melhor dos melhores

Assim como todo código apresentado neste livro, você encontra as versões modificadas do código-fonte em: https://github.com/logicadojogo/fontes/tree/desafios.

10.2 Use a caixa de ferramentas para criar seus próprios jogos

Durante os capítulos, construímos um código base para auxiliar no desenvolvimento dos jogos. Embora ele tenha sido visto como parte integral de cada jogo e mesmo sendo muito básico, você pode usá-lo como uma biblioteca à parte em seus projetos, melhorando o código existente e contribuindo com novos métodos auxiliares: https://github.com/logicadojogo/fontes/tree/master/base.

Capítulo 11

Referências bibliográficas

DEITEL, H. M. *Java*: como programar. 6. ed. Trad. Edson Furmankiewicz. São Paulo: Pearson Prentice Hall, 2005.

HARBOUR, Jonathan S. *Programação de games com Java*. Trad. Carlos Eduardo Santi. São Paulo: Cengage Learning, 2010.

KENT, Steven L. *The ultimate history of video games*: from Pong to Pokemon - The story behind the craze that touched our lives and changed the world. New York: Three Rivers Press, 2001.

MOOT, Tony. *1001 videogames para jogar antes de morrer*. Trad. Livia de Almeida. Rio de Janeiro: Sextante, 2013.

NOVAK, Jeannie. *Desenvolvimento de games*. Trad. Pedro Cesar de Conti. São Paulo: Cengage Learning, 2010.