Java 2D games programming

Jan Bodnar

October 22, 2019

Contents

Preface

This is Java 2D games programming e-book. It shows how to create six simple 2D games in Java.

The e-book contains the following chapters:

1. Foundations
2. Snake
3. Tetris
4. Minesweeper
5. Balloons
6. Flappy bird
7. Cannon

In the Foundations chapter, we quickly mention some fundamental aspects of game programming in Java Swing, including the paint mechanism, graphics object, and frame rate.

In the Snake game, we move a snake and try to eat as many apples as possible, while avoiding boundaries and the snake body.

Tetris is a falling block puzzle game. The objective is to form rows from the pieces by rotating them. In the Minesweeper game, we try to find all mines in a minefield. The player uses information about neighbouring mines to find out where the mines might be located.

In the Balloons game, we have to prevent balloons from reaching the top of the window. We control the gun sight to crack them. In the Flappy bird game we control a bird and try to avoid pipes on the journey. In the Cannon game, we operate a cannon to shoot incoming enemy planes.

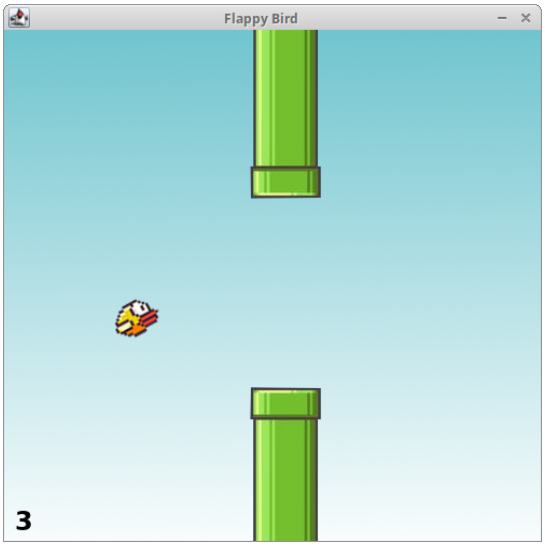

Figure 1: Flappy bird

About the author

My name is Jan Bodnar. I am Hungarian and I come from Slovakia. Currently I live in Bratislava. Apart from working with computers I read a lot. Mostly belles-letters and history. I study several foreign languages. Being aware of that healthy soul lives in a healthy body, I exercise regularly, work in the garden, and often take tours. I practice Tai Chi and love dancing.

I run the zetcode.com site. *Java 2D games programming* is my ninth e-book. I have also written the following e-books: *Swing layout management*,*MySQL Java programming*, *Advanced PyQt5*, *Advanced wxPython*, *SQLite Python*, and *Advanced Java Swing*, *Tkinter programming*, and *Introduction to Windows API programming*. I hope this e-book will be useful to you.

Thank you for buying this e-book.

Chapter 1

Foundations

In this chapter we talk about some foundations of game development in Java and Swing. These are common to all the games in this e-book. We recommend to read the `Java 2D tutorial` to get the basics of 2D graphics in Java.

1.1 Focus

Many Swing components can be operated with the keyboard. For a key press to affect a component, the component must have the keyboard focus.

```
setFocusable(true);
```

For games that are controlled with keyboard, the board must have focus in order to receive key events. The focus is set with the `setFocusable()` method.

1.2 ImageIcon

`Icon` is small fixed size picture, typically used to decorate components. `ImageIcon` is an implementation of the `Icon` interface that paints icons from images. Images can be created from a URL, filename, or byte array. `ImageIcon` is used to load images in our games.

```
private Image head;
private Image body;

private void loadImages() {

    var iih = new ImageIcon("src/resources/head.png");
    head = iih.getImage();

    var iid = new ImageIcon("src/resources/dot.png");
    body = iid.getImage();
}
```

PNG images are loaded from the resources. We get the `Images` from the icon with `getImage()`.

```
private void drawSnake(Graphics g) {
```

```
    for (int z = 0; z < snake.size(); z++) {

        if (z == 0) {

            g.drawImage(snake.getHead(), p[z].x, p[z].y, this);
        } else {

            g.drawImage(snake.getBody(), p[z].x, p[z].y, this);
        }
    }
}
```

Later, the `Images` are drawn with `drawImage()`.

1.3 Synchronization

Some windowing systems do buffering of graphics systems. These need to be synchronized for smooth animations.

```
Toolkit.getDefaultToolkit().sync();
```

This method is needed for Linux in order to have smooth animations. On Linux, the display buffer is not automatically flushed; therefore, we need to call the `sync()` method.

1.4 Frame rate

The *frame rate* is the number of frames or images that are displayed in a film, TV, or computer game. The greater the frame rate, the smoother the animation appears. Thirty frames per second is a rate that is generally considered OK for many computer games.

Also note that setting the frame rate above 60 FPS may not have any effect if the monitor has 60 Hz refresh rate. (Today, many screens have 60 Hz refresh rate.)

```
private final int FPS = 60;
private final int PERIOD = 1000 / FPS;
```

In the Snake game, we choose sixty frames per second frame rate. We calculate the period from the FPS. The granularity of the timer is in milliseconds; therefore, we have the following formula: `PERIOD = 1000 / FPS`.

```
timer = new Timer(PERIOD, new GameCycle());
timer.start();
```

Later the timer is executed at the calculated period.

1.5 Random number generation

The `Random` class is preferred to the `Math.random()` method. The `Random` class is more efficient and less biased.[1]

```
int r = random.nextInt(RAND_POS);
```

In the Snake game, we use the `nextInt()` method of the `Random` class to generate random integers.

1.6 The paint mechanism

The custom painting code should be placed in the `paintComponent()` method. This method is invoked when it is time to paint. The paint subsystem first calls the `paint()` method. This method invokes the following three methods:

- `paintComponent()`
- `paintBorder()`
- `paintChildren()`

Sometimes we might want to override the `paintBorder()` or the `paintChildren()` methods. In most cases, we override the `paintComponent()` method.

```
@Override
public void paintComponent(Graphics g) {

    super.paintComponent(g);
    doDrawing(g);
}
```

Custom painting is performed inside the `paintComponent()` method, which we override. The `super.paintComponent()` method calls the method of the parent class. It does some necessary work to prepare a component for drawing. We delegate the drawing to the custom `doDrawing()` method.

The program invokes `repaint()` on the component, which registers an asynchronous request to the AWT that this component needs to be repainted.

1.7 The Graphics object

The `paintComponent's` sole parameter is a `Graphics` object. It exposes a number of methods for drawing 2D shapes and obtaining information about the application's graphics environment. The `Graphics2D` class extends the `Graphics` class to provide more sophisticated control over geometry, coordinate transformations, color management, and text layout.

The `Graphics` object is initialized before it is passed to the `paintComponent()` method, and then it is turned over to the `paintBorder()` and `paintChildren()` methods. This reuse improves performance but it may lead to problems if the painting code permanently changes the `Graphics` state. Therefore, we must

[1] https://community.oracle.com/message/6596485#thread-message-6596485

either restore the original settings or work with a copy of the `Graphics` object. The copy is created with the `Graphics's create()` method; it must be later released with the `dispose()` method.

In practical terms, the copy of the `Graphics` object does not need to be created if we set the following properties: font, colour, and rendering hints. For all other properties, (especially clip, composite operations, and transformations), we must create a copy of the `Graphics` object and later dispose it.

1.8 Rendering hints

Rendering hints are graphics attributes that control the quality of the rendering output. In the games we use rendering hints to get smooth lines and fonts.

```
var g2d = (Graphics2D) g;

g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
        RenderingHints.VALUE_ANTIALIAS_ON);

g2d.setRenderingHint(RenderingHints.KEY_RENDERING,
        RenderingHints.VALUE_RENDER_QUALITY);
```

The `Graphics2D` class is a fundamental class for rendering graphics in Java 2D. It represents number of devices in a generic way. It extends the older `Graphics` object. This casting is necessary to get access to all advanced operations, including rendering hints.

1.9 The pack method

The games have a fixed size. However, various desktop environments have different window decorations. If we want to have portable games, we cannot simply set the frame size with the `setSize()` method.

Instead of using `setSize()`, we set the preferred size of the board component with the `setPreferredSize()` and call the frame's `pack()` method. The `pack()` method sizes the frame so that all its contents are at or above their preferred size values.

Chapter 2

Snake

Snake is an older classic video game. It was first created in late 70s. Later it was brought to PCs. In this game the player controls a snake. The objective is to eat as many apples as possible. Each time the snake eats an apple its body grows. The snake must avoid the walls and its own body. This game is sometimes called Nibbles.

The size of each of the joints of a snake is 10 px. The snake is controlled with the cursor keys. Initially, the snake has three joints. If the game is finished, the "Game over" message is displayed in the middle of the board.

Listing 2.1: Direction.java

```
package com.zetcode;

public enum Direction {
    RIGHT,
    LEFT,
    UP,
    DOWN
}
```

The `Direction` enumeration defines the direction of the snake's movement.

Listing 2.2: Apple.java

```
package com.zetcode;

import java.awt.Image;
import javax.swing.ImageIcon;

public class Apple {

    private int x;
    private int y;
    private Image apple;

    public Apple() {

        loadImage();
    }
```

```
    private void loadImage() {

        var iia = new ImageIcon("src/resources/apple.png");
        apple = iia.getImage();
    }

    public int getX() {

        return x;
    }

    public void setX(int x) {

        this.x = x;
    }

    public int getY() {

        return y;
    }

    public void setY(int y) {

        this.y = y;
    }

    public Image getApple() {

        return apple;
    }

    public void setApple(Image apple) {

        this.apple = apple;
    }
}
```

The `Apple` class represents the apple sprite. It contains attributes and methods for its image and coordinates.

Listing 2.3: Snake.java

```
package com.zetcode;

import java.awt.Image;
import javax.swing.ImageIcon;

public class Snake {

    private Image head;
    private Image body;
    private Direction direction = Direction.RIGHT;

    private int nOfDots = 3;

    public Snake() {

        initSnake();
    }
```

```
    private void initSnake() {

        loadImages();
    }

    private void loadImages() {

        var iih = new ImageIcon("src/resources/head.png");
        head = iih.getImage();

        var iid = new ImageIcon("src/resources/dot.png");
        body = iid.getImage();
    }

    public void grow() {

        nOfDots++;
    }

    public Image getHead() {

        return head;
    }

    public Image getBody() {

        return body;
    }

    public int size() {

        return nOfDots;
    }

    public Direction getDirection() {

        return direction;
    }

    public void setDirection(Direction direction) {

        this.direction = direction;
    }
}
```

The `Snake` class represents the snake object. It consists of two images: a head and a body part. We store the number of body parts and the direction of the movement.

```
public void grow() {

    nOfDots++;
}
```

The `grow()` method is called when the snake eats an apple. The `nOfDots` variable stores the length of the snake's body.

Listing 2.4: Animation.java

```
package com.zetcode;

public class Animation {

    private final int NUMBER_OF_ANIM_STEPS = 10;
    private int steps = 0;

    public void inc() {

        steps++;
    }

    public void reset() {

        steps = 0;
    }

    public boolean finished() {

        return steps == NUMBER_OF_ANIM_STEPS;
    }
}
```

The `Animation` class represents the animation in the game. The snake moves smoothly on a pixel by pixel basis. The animation consists of ten steps, which corresponds to the body of the snake, whose dots are 10x10 px.

Listing 2.5: Board.java

```
package com.zetcode;

import javax.swing.JPanel;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Point;
import java.awt.RenderingHints;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.util.Random;
import javax.swing.Timer;

class MyPoint extends Point {

    public MyPoint(int x, int y) {
        super(x, y);
    }

    public int deltaX = 0;
    public int deltaY = 0;
}

public class Board extends JPanel {

    private final int BOARD_WIDTH = 300;
```

```
    private final int BOARD_HEIGHT = 300;
    private final int DOT_SIZE = 10;
    private final int ALL_DOTS = 900;
    private final int RAND_POS = 29;

    private final int FPS = 60;
    private final int PERIOD = 1000 / FPS;

    private final MyPoint p[] = new MyPoint[ALL_DOTS];

    private boolean inGame = true;

    private Timer timer;

    private Snake snake;
    private Apple apple;

    private Animation anim;

    public Board() {

        initBoard();
    }

    private void initBoard() {

        addKeyListener(new TAdapter());
        setBackground(Color.black);
        setFocusable(true);

        setPreferredSize(new Dimension(BOARD_WIDTH, BOARD_HEIGHT));
        initGame();
    }

    private void initGame() {

        snake = new Snake();
        apple = new Apple();
        anim = new Animation();

        for (int y = 0; y < ALL_DOTS; y++) {

            p[y] = new MyPoint(0, 0);
        }

        for (int z = 0; z < snake.size(); z++) {

            p[z] = new MyPoint(50 - z * DOT_SIZE, 50);
        }

        locateApple();

        timer = new Timer(PERIOD, new GameCycle());
        timer.start();
    }

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

        doDrawing(g);
    }
```

```
    private void doDrawing(Graphics g) {

        if (inGame) {

            drawApple(g);
            drawSnake(g);

            Toolkit.getDefaultToolkit().sync();

        } else {

            gameOver(g);
        }
    }

    private void drawSnake(Graphics g) {

        for (int z = 0; z < snake.size(); z++) {

            if (z == 0) {

                g.drawImage(snake.getHead(), p[z].x, p[z].y, this);
            } else {

                g.drawImage(snake.getBody(), p[z].x, p[z].y, this);
            }
        }
    }

    private void drawApple(Graphics g) {

        g.drawImage(apple.getApple(), apple.getX(),
                    apple.getY(), this);
    }

    private void gameOver(Graphics g) {

        var g2d = (Graphics2D) g;

        g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                RenderingHints.VALUE_ANTIALIAS_ON);

        g2d.setRenderingHint(RenderingHints.KEY_RENDERING,
                RenderingHints.VALUE_RENDER_QUALITY);

        var msg = "Game over";
        var smallFont = new Font("Helvetica", Font.BOLD, 14);
        var fontMetrics = getFontMetrics(smallFont);

        g2d.setColor(Color.white);
        g2d.setFont(smallFont);
        g2d.drawString(msg, (BOARD_WIDTH -
                fontMetrics.stringWidth(msg)) / 2, BOARD_HEIGHT / 2);
    }

    private void checkApple() {

        if (p[0].x == apple.getX() && p[0].y == apple.getY()) {

            Point last = p[snake.size() - 1];
```

```
            if (snake.getDirection() == Direction.RIGHT
                    || snake.getDirection() == Direction.LEFT) {

                p[snake.size()].x = last.x;
                p[snake.size()].y = last.y - DOT_SIZE;

            } else {

                p[snake.size()].x = last.x - DOT_SIZE;;
                p[snake.size()].y = last.y;
            }

            snake.grow();
            locateApple();
        }
    }

    private void move() {

        if (anim.finished()) {

            anim.reset();

            if (snake.getDirection() == Direction.RIGHT) {

                p[0].deltaX = 1;
                p[0].deltaY = 0;
            }

            if (snake.getDirection() == Direction.LEFT) {

                p[0].deltaX = -1;
                p[0].deltaY = 0;
            }

            if (snake.getDirection() == Direction.UP) {

                p[0].deltaX = 0;
                p[0].deltaY = -1;
            }

            if (snake.getDirection() == Direction.DOWN) {

                p[0].deltaX = 0;
                p[0].deltaY = 1;
            }

            for (int z = 0; z < snake.size(); z++) {

                if (p[z + 1].x == p[z].x) {

                    p[z + 1].deltaX = 0;
                } else if (p[z + 1].x > p[z].x) {

                    p[z + 1].deltaX = -1;
                } else {

                    p[z + 1].deltaX = 1;
                }

                if (p[z + 1].y == p[z].y) {
```

```
                p[z + 1].deltaY = 0;
            } else if (p[z + 1].y > p[z].y) {

                p[z + 1].deltaY = -1;
            } else {

                p[z + 1].deltaY = 1;
            }
        }

    } else {

        anim.inc();

        for (int z = 0; z < snake.size(); z++) {

            p[z].translate(p[z].deltaX, p[z].deltaY);
        }
    }
}

private void checkCollision() {

    for (int z = snake.size(); z > 0; z--) {

        if (z > 4 && p[0].x == p[z].x && p[0].y == p[z].y) {

            inGame = false;
        }
    }

    if (p[0].y >= BOARD_HEIGHT) {

        inGame = false;
    }

    if (p[0].y < 0) {

        inGame = false;
    }

    if (p[0].x >= BOARD_WIDTH) {

        inGame = false;
    }

    if (p[0].x < 0) {

        inGame = false;
    }

    if (!inGame) {

        timer.stop();
    }
}

private void locateApple() {

    var random = new Random();

    int r = random.nextInt(RAND_POS);
```

```
        apple.setX(r * DOT_SIZE);

        r = random.nextInt(RAND_POS);
        apple.setY(r * DOT_SIZE);
    }

    private void doGameCycle() {

        if (inGame) {

            checkApple();
            checkCollision();
            move();
        }

        repaint();
    }

    private class TAdapter extends KeyAdapter {

        @Override
        public void keyPressed(KeyEvent e) {

            int key = e.getKeyCode();

            if (key == KeyEvent.VK_LEFT &&

                    snake.getDirection() != Direction.RIGHT) {

                snake.setDirection(Direction.LEFT);
            }

            if (key == KeyEvent.VK_RIGHT &&
                    snake.getDirection() != Direction.LEFT) {

                snake.setDirection(Direction.RIGHT);
            }

            if (key == KeyEvent.VK_UP &&
                    snake.getDirection() != Direction.DOWN) {

                snake.setDirection(Direction.UP);
            }

            if (key == KeyEvent.VK_DOWN &&
                    snake.getDirection() != Direction.UP) {

                snake.setDirection(Direction.DOWN);
            }
        }
    }

    private class GameCycle implements ActionListener {

        @Override
        public void actionPerformed(ActionEvent e) {

            doGameCycle();
        }
    }
}
```

The board contains the logic of the game.

```
class MyPoint extends Point {

    public MyPoint(int x, int y) {
        super(x, y);
    }

    public int deltaX = 0;
    public int deltaY = 0;
}
```

The `MyPoint` class stores the `x` and `y` coordinates of the snake part and the `deltaX` and `deltaY` movement variables.

```
private final int BOARD_WIDTH = 300;
private final int BOARD_HEIGHT = 300;
private final int DOT_SIZE = 10;
private final int ALL_DOTS = 900;
private final int RAND_POS = 29;
```

The `BOARD_WIDTH` and `BOARD_HEIGHT` constants determine the size of the board. The `DOT_SIZE` is the size of the apple and the dot of the snake. The `ALL_DOTS` constant defines the maximum number of possible dots on the board (900 = (300*300)/(10*10)). The `RAND_POS` constant is used to calculate a random position for an apple.

```
private final int FPS = 60;
private final int PERIOD = 1000 / FPS;
```

The game speed is sixty frames per second.

```
private final MyPoint p[] = new MyPoint[ALL_DOTS];
```

The `p` array stores the x and y coordinates of all joints of a snake.

```
private void initGame() {

    snake = new Snake();
    apple = new Apple();
    anim = new Animation();
...
```

Inside the `initGame()` method, we first create the snake, apple, and animation objects.

```
for (int y = 0; y < ALL_DOTS; y++) {

    p[y] = new MyPoint(0, 0);
}
```

We initialize the `p` array.

```
for (int z = 0; z < snake.size(); z++) {

    p[z] = new MyPoint(50 - z * DOT_SIZE, 50);
}
```

We initialize the snake. At the beginning, the snake has a head and two joints.

```
locateApple();
```

We randomly position an apple on the board.

```
timer = new Timer(PERIOD, new GameCycle());
timer.start();
```

We finish the game initialization by creating a timer.

```
private void drawSnake(Graphics g) {

    for (int z = 0; z < snake.size(); z++) {

        if (z == 0) {

            g.drawImage(snake.getHead(), p[z].x, p[z].y, this);
        } else {

            g.drawImage(snake.getBody(), p[z].x, p[z].y, this);
        }
    }
}
```

The drawing of the snake consists of two parts. We draw its head and the body parts.

```
private void checkApple() {

    if (p[0].x == apple.getX() && p[0].y == apple.getY()) {
...
```

In the `checkApple()` method, we check if the snake's head hits the apple. In the if statement, we compare the coordinates of the head and the apple.

```
Point last = p[snake.size() - 1];
```

We get the coordinates of the last snake joint. The snake grows at its tail.

```
if (snake.getDirection() == Direction.RIGHT
      || snake.getDirection() == Direction.LEFT) {

    p[snake.size()].x = last.x;
    p[snake.size()].y = last.y - DOT_SIZE;

} else {

    p[snake.size()].x = last.x - DOT_SIZE;;
    p[snake.size()].y = last.y;
}
```

Depending on the snake movement, we genereate a new `MyPoint` in the `p` array.

```
snake.grow();
locateApple();
```

The snake grows and we generate a new apple on the board.

```
private void move() {
...
```

The `move()` method is responsible for the movement of the snake.

```
if (anim.finished()) {

    anim.reset();

    if (snake.getDirection() == Direction.RIGHT) {

        p[0].deltaX = 1;
        p[0].deltaY = 0;
    }
...
```

The animation object animates the snake sprite. If it has finished, we reset it and update the `deltaX` and `deltaY` variables of the head according to the latest snake direction.

```
for (int z = 0; z < snake.size(); z++) {

    if (p[z + 1].x == p[z].x) {

        p[z + 1].deltaX = 0;
    } else if (p[z + 1].x > p[z].x) {

        p[z + 1].deltaX = -1;
    } else {

        p[z + 1].deltaX = 1;
    }

    if (p[z + 1].y == p[z].y) {

        p[z + 1].deltaY = 0;
    } else if (p[z + 1].y > p[z].y) {

        p[z + 1].deltaY = -1;
    } else {

        p[z + 1].deltaY = 1;
    }
}
```

We set the `deltaX` and `deltaY` variables for all the remaining joints of the snake. The delta variables of each of the joints depend on the predecessor joint—it takes its direction from its predecessor.

```
} else {

    anim.inc();

    for (int z = 0; z < snake.size(); z++) {

        p[z].translate(p[z].deltaX, p[z].deltaY);
    }
}
```

During the animation, we increase the counter and translate the coordinates.

```
for (int z = snake.size(); z > 0; z--) {

    if (z > 4 && p[0].x == p[z].x && p[0].y == p[z].y) {

        inGame = false;
    }
}
```

If the snake hits one of its joints with its head the game is over. The `inGame` variable is set to `false`.

```
if (p[0].y >= BOARD_HEIGHT) {

    inGame = false;
}
```

If the snake's head hits the bottom of the board, the game is finished.

```
if (!inGame) {

    timer.stop();
}
```

When the `inGame` is set to `false`, the timer is stopped.

```
private void locateApple() {

    var random = new Random();

    int r = random.nextInt(RAND_POS);
    apple.setX(r * DOT_SIZE);

    r = random.nextInt(RAND_POS);
    apple.setY(r * DOT_SIZE);
}
```

The `locateApple()` randomly positions the apple object on the board.

```
@Override
public void keyPressed(KeyEvent e) {

    int key = e.getKeyCode();

    if (key == KeyEvent.VK_LEFT &&

            snake.getDirection() != Direction.RIGHT) {

        snake.setDirection(Direction.LEFT);
    }
...
```

The game is controlled with the cursors keys. The keys set the direction of the snake with `setDirection()`.

```
private class GameCycle implements ActionListener {

    @Override
    public void actionPerformed(ActionEvent e) {

        doGameCycle();
```

```
    }
}
```

The `ActionListener` periodically runs the `doGameCycle()` method.

Listing 2.6: SnakeGame.java

```
package com.zetcode;

import java.awt.EventQueue;
import javax.swing.JFrame;

/**
 * Java Snake game
 *
 * Author: Jan Bodnar
 * Website: http://zetcode.com
 */

public class SnakeGame extends JFrame {

    public SnakeGame() {

        initUI();
    }

    private void initUI() {

        add(new Board());

        setResizable(false);
        pack();

        setTitle("Snake");
        setLocationRelativeTo(null);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }


    public static void main(String[] args) {

        EventQueue.invokeLater(() -> {

            var game = new SnakeGame();
            game.setVisible(true);
        });
    }
}
```

The `SnakeGame` sets up the Snake game.

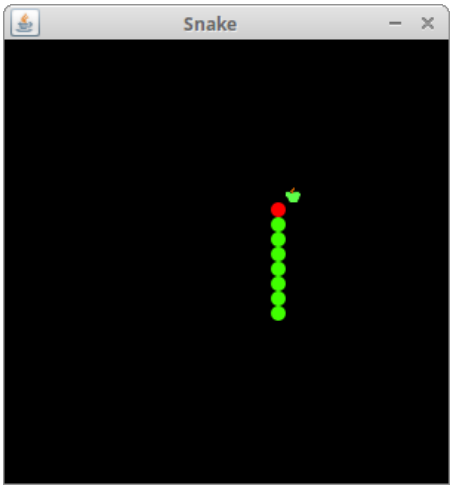

Figure 2.1: Snake

Chapter 3

Tetris

Tetris is one of the most popular computer games ever created. The original game was designed and programmed by a Russian programmer Alexey Pajitnov in 1985. Since then, Tetris is available on almost every computer platform in lots of variations.

Tetris is called a falling block puzzle game. In this game, we have seven different shapes called tetrominoes: S-shape, Z-shape, T-shape, L-shape, Line-shape, MirroredL-shape, and a Square-shape. Each of these shapes is formed with four squares. The shapes are falling down the board. The object of the Tetris game is to move and rotate the shapes so that they fit as much as possible. If we manage to form a row, the row is destroyed and we score. We play the Tetris game until we top out.

Figure 3.1: Tetrominoes

The tetrominoes are drawn using the Swing painting API. We use the `Timer` to create a game cycle. The shapes move on a square by square basis (not pixel by pixel). Mathematically the board in the game is a simple list of numbers.

The game starts immediately after it is launched. We can pause the game by pressing the `P` key. The `Space` key drops the Tetris piece immediately to the bottom. The `D` key drops the piece one line down. (It can be used to speed up the falling a bit.) The game goes at constant speed, no acceleration is implemented. The score is the number of lines that we have removed.

Listing 3.1: Shape.java

```
package com.zetcode;

import java.util.Random;
```

```
public class Shape {

    protected enum Tetrominoe { NoShape, ZShape, SShape, LineShape,
        TShape, SquareShape, LShape, MirroredLShape }

    private Tetrominoe pieceShape;
    private int coords[][];
    private int[][][] coordsTable;


    public Shape() {

        initShape();
    }

    private void initShape() {

        coords = new int[4][2];

        coordsTable = new int[][][] {
                { { 0, 0 },   { 0, 0 },   { 0, 0 },   { 0, 0 } },
                { { 0, -1 },  { 0, 0 },   { -1, 0 },  { -1, 1 } },
                { { 0, -1 },  { 0, 0 },   { 1, 0 },   { 1, 1 } },
                { { 0, -1 },  { 0, 0 },   { 0, 1 },   { 0, 2 } },
                { { -1, 0 },  { 0, 0 },   { 1, 0 },   { 0, 1 } },
                { { 0, 0 },   { 1, 0 },   { 0, 1 },   { 1, 1 } },
                { { -1, -1 }, { 0, -1 },  { 0, 0 },   { 0, 1 } },
                { { 1, -1 },  { 0, -1 },  { 0, 0 },   { 0, 1 } }
        };

        setShape(Tetrominoe.NoShape);
    }

    protected void setShape(Tetrominoe shape) {

        for (int i = 0; i < 4 ; i++) {

            for (int j = 0; j < 2; ++j) {

                coords[i][j] = coordsTable[shape.ordinal()][i][j];
            }
        }

        pieceShape = shape;
    }

    private void setX(int index, int x) { coords[index][0] = x; }
    private void setY(int index, int y) { coords[index][1] = y; }
    public int x(int index) { return coords[index][0]; }
    public int y(int index) { return coords[index][1]; }
    public Tetrominoe getShape()  { return pieceShape; }

    public void setRandomShape() {

        var r = new Random();
        int x = Math.abs(r.nextInt()) % 7 + 1;

        Tetrominoe[] values = Tetrominoe.values();
        setShape(values[x]);
    }

    public int minX() {
```

```
        int m = coords[0][0];

        for (int i=0; i < 4; i++) {

            m = Math.min(m, coords[i][0]);
        }

        return m;
    }


    public int minY() {

        int m = coords[0][1];

        for (int i=0; i < 4; i++) {

            m = Math.min(m, coords[i][1]);
        }

        return m;
    }

    public Shape rotateLeft() {

        if (pieceShape == Tetrominoe.SquareShape) {

            return this;
        }

        var result = new Shape();
        result.pieceShape = pieceShape;

        for (int i = 0; i < 4; ++i) {

            result.setX(i, y(i));
            result.setY(i, -x(i));
        }

        return result;
    }

    public Shape rotateRight() {

        if (pieceShape == Tetrominoe.SquareShape) {

            return this;
        }

        var result = new Shape();
        result.pieceShape = pieceShape;

        for (int i = 0; i < 4; ++i) {

            result.setX(i, -y(i));
            result.setY(i, x(i));
        }

        return result;
    }
}
```

The `Shape` class provides information about a Tetris piece.

```
protected enum Tetrominoe { NoShape, ZShape, SShape, LineShape,
            TShape, SquareShape, LShape, MirroredLShape }
```

The `Tetrominoe` enum holds seven Tetris shape names and the empty shape called `NoShape`.

```
private void initShape() {

    coords = new int[4][2];

    coordsTable = new int[][][] {
            { { 0, 0 },   { 0, 0 },   { 0, 0 },   { 0, 0 } },
            { { 0, -1 },  { 0, 0 },   { -1, 0 },  { -1, 1 } },
            { { 0, -1 },  { 0, 0 },   { 1, 0 },   { 1, 1 } },
            { { 0, -1 },  { 0, 0 },   { 0, 1 },   { 0, 2 } },
            { { -1, 0 },  { 0, 0 },   { 1, 0 },   { 0, 1 } },
            { { 0, 0 },   { 1, 0 },   { 0, 1 },   { 1, 1 } },
            { { -1, -1 }, { 0, -1 },  { 0, 0 },   { 0, 1 } },
            { { 1, -1 },  { 0, -1 },  { 0, 0 },   { 0, 1 } }
    };

    setShape(Tetrominoe.NoShape);
}
```

The `coords` array holds the actual coordinates of a Tetris piece. The `coordsTable` array holds all possible coordinate values of the Tetris pieces. This is a template from which all pieces take their coordinate values.

```
protected void setShape(Tetrominoe shape) {

    for (int i = 0; i < 4 ; i++) {

        for (int j = 0; j < 2; ++j) {

            coords[i][j] = coordsTable[shape.ordinal()][i][j];
        }
    }

    pieceShape = shape;
}
```

In these for loops we put one row of the coordinate values from the `coordsTable` into the `coords` array of a Tetris piece. Note the usage of the `ordinal()` method. In C++, an enum type is essentially an integer. Unlike in C++, Java enums are full classes and the `ordinal()` method returns the current position of the enum type in the enum object.

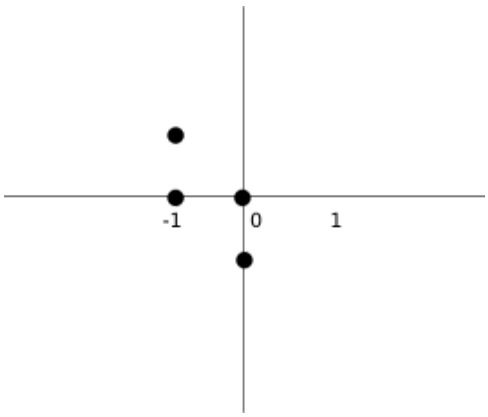

Figure 3.2: Coordinates

Figure 3.2 helps understand the coordinate values a bit more. The `coords` array saves the coordinates of the Tetris piece. The following diagram illustrates the rotated S-shape. It has the following coordinates: (-1, 1), (-1, 0), (0, 0), and (0, -1).

```
public Shape rotateLeft() {

    if (pieceShape == Tetrominoe.SquareShape) {

        return this;
    }

    var result = new Shape();
    result.pieceShape = pieceShape;

    for (int i = 0; i < 4; ++i) {

        result.setX(i, y(i));
        result.setY(i, -x(i));
    }

    return result;
}
```

This code rotates a piece to the left. The square does not have to be rotated. That's why we simply return the reference to the current object. Looking at the previous image will help to understand the rotation.

Listing 3.2: Board.java

```
package com.zetcode;

import com.zetcode.Shape.Tetrominoe;

import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.Timer;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
```

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;

public class Board extends JPanel {

    private final int BOARD_WIDTH = 10;
    private final int BOARD_HEIGHT = 22;
    private final int PERIOD = 300;

    private Timer timer;
    private boolean isFallingFinished = false;
    private boolean isPaused = false;
    private int numLinesRemoved = 0;
    private int curX = 0;
    private int curY = 0;

    private JLabel statusbar;
    private Shape curPiece;
    private Tetrominoe[] board;

    public Board(Tetris parent) {

        initBoard(parent);
    }

    private void initBoard(Tetris parent) {

        setPreferredSize(new Dimension(200, 400));
        setFocusable(true);

        statusbar = parent.getStatusBar();
        addKeyListener(new TAdapter());
    }

    private int squareWidth() {

        return (int) getSize().getWidth() / BOARD_WIDTH;
    }

    private int squareHeight() {

        return (int) getSize().getHeight() / BOARD_HEIGHT;
    }

    private Tetrominoe shapeAt(int x, int y) {

        return board[(y * BOARD_WIDTH) + x];
    }

    void start() {

        curPiece = new Shape();
        board = new Tetrominoe[BOARD_WIDTH * BOARD_HEIGHT];

        clearBoard();
        newPiece();

        timer = new Timer(PERIOD, new GameCycle());
        timer.start();
    }
```

```
    private void pause() {

        isPaused = !isPaused;

        if (isPaused) {

            statusbar.setText("paused");
        } else {

            statusbar.setText(String.valueOf(numLinesRemoved));
        }

        repaint();
    }

    @Override
    public void paintComponent(Graphics g) {

        super.paintComponent(g);
        doDrawing(g);
    }

    private void doDrawing(Graphics g) {

        var size = getSize();
        int boardTop = (int) size.getHeight()
                - BOARD_HEIGHT * squareHeight();

        for (int i = 0; i < BOARD_HEIGHT; i++) {

            for (int j = 0; j < BOARD_WIDTH; j++) {

                Tetrominoe shape = shapeAt(j, BOARD_HEIGHT - i - 1);

                if (shape != Tetrominoe.NoShape) {

                    drawSquare(g, j * squareWidth(),
                            boardTop + i * squareHeight(), shape);
                }
            }
        }

        if (curPiece.getShape() != Tetrominoe.NoShape) {

            for (int i = 0; i < 4; i++) {

                int x = curX + curPiece.x(i);
                int y = curY - curPiece.y(i);

                drawSquare(g, x * squareWidth(),
                    boardTop + (BOARD_HEIGHT - y - 1) * squareHeight(),
                    curPiece.getShape());
            }
        }
    }

    private void dropDown() {

        int newY = curY;

        while (newY > 0) {
```

```
            if (!tryMove(curPiece, curX, newY - 1)) {

                break;
            }

            newY--;
        }

        pieceDropped();
    }

    private void oneLineDown() {

        if (!tryMove(curPiece, curX, curY - 1)) {

            pieceDropped();
        }
    }

    private void clearBoard() {

        for (int i = 0; i < BOARD_HEIGHT * BOARD_WIDTH; i++) {

            board[i] = Tetrominoe.NoShape;
        }
    }

    private void pieceDropped() {

        for (int i = 0; i < 4; i++) {

            int x = curX + curPiece.x(i);
            int y = curY - curPiece.y(i);
            board[(y * BOARD_WIDTH) + x] = curPiece.getShape();
        }

        removeFullLines();

        if (!isFallingFinished) {

            newPiece();
        }
    }

    private void newPiece() {

        curPiece.setRandomShape();
        curX = BOARD_WIDTH / 2 + 1;
        curY = BOARD_HEIGHT - 1 + curPiece.minY();

        if (!tryMove(curPiece, curX, curY)) {

            curPiece.setShape(Tetrominoe.NoShape);
            timer.stop();

            var msg = String.format("Game over. Score: %d",
                    numLinesRemoved);
            statusbar.setText(msg);
        }
    }
```

```
    private boolean tryMove(Shape newPiece, int newX, int newY) {

        for (int i = 0; i < 4; i++) {

            int x = newX + newPiece.x(i);
            int y = newY - newPiece.y(i);

            if (x < 0 || x >= BOARD_WIDTH || y < 0
                    || y >= BOARD_HEIGHT) {

                return false;
            }

            if (shapeAt(x, y) != Tetrominoe.NoShape) {

                return false;
            }
        }

        curPiece = newPiece;
        curX = newX;
        curY = newY;

        repaint();

        return true;
    }

    private void removeFullLines() {

        int numFullLines = 0;

        for (int i = BOARD_HEIGHT - 1; i >= 0; i--) {

            boolean lineIsFull = true;

            for (int j = 0; j < BOARD_WIDTH; j++) {

                if (shapeAt(j, i) == Tetrominoe.NoShape) {

                    lineIsFull = false;
                    break;
                }
            }

            if (lineIsFull) {

                numFullLines++;

                for (int k = i; k < BOARD_HEIGHT - 1; k++) {
                    for (int j = 0; j < BOARD_WIDTH; j++) {
                        board[(k * BOARD_WIDTH) + j] =
                            shapeAt(j, k + 1);
                    }
                }
            }
        }

        if (numFullLines > 0) {

            numLinesRemoved += numFullLines;
```

```
            statusbar.setText(String.valueOf(numLinesRemoved));
            isFallingFinished = true;
            curPiece.setShape(Tetrominoe.NoShape);
        }
    }

    private void drawSquare(Graphics g, int x, int y,
                            Tetrominoe shape) {

        Color colors[] = {new Color(0, 0, 0), new Color(204, 102, 102),
                new Color(102, 204, 102), new Color(102, 102, 204),
                new Color(204, 204, 102), new Color(204, 102, 204),
                new Color(102, 204, 204), new Color(218, 170, 0)
        };

        var color = colors[shape.ordinal()];

        g.setColor(color);
        g.fillRect(x + 1, y + 1, squareWidth() - 2,
            squareHeight() - 2);

        g.setColor(color.brighter());
        g.drawLine(x, y + squareHeight() - 1, x, y);
        g.drawLine(x, y, x + squareWidth() - 1, y);

        g.setColor(color.darker());
        g.drawLine(x + 1, y + squareHeight() - 1,
                x + squareWidth() - 1, y + squareHeight() - 1);
        g.drawLine(x + squareWidth() - 1, y + squareHeight() - 1,
                x + squareWidth() - 1, y + 1);
    }

    private void doGameCycle() {

        update();
        repaint();
    }

    private void update() {

        if (isPaused) {

            return;
        }

        if (isFallingFinished) {

            isFallingFinished = false;
            newPiece();
        } else {

            oneLineDown();
        }
    }

    class TAdapter extends KeyAdapter {

        @Override
        public void keyPressed(KeyEvent e) {


            if (curPiece.getShape() == Tetrominoe.NoShape) {
```

```
                return;
            }

            int keycode = e.getKeyCode();


            switch (keycode) {

                case KeyEvent.VK_P:
                    pause();
                    break;

                case KeyEvent.VK_LEFT:
                    tryMove(curPiece, curX - 1, curY);
                    break;

                case KeyEvent.VK_RIGHT:
                    tryMove(curPiece, curX + 1, curY);
                    break;

                case KeyEvent.VK_DOWN:
                    tryMove(curPiece.rotateRight(), curX, curY);
                    break;

                case KeyEvent.VK_UP:
                    tryMove(curPiece.rotateLeft(), curX, curY);
                    break;

                case KeyEvent.VK_SPACE:
                    dropDown();
                    break;

                case KeyEvent.VK_D:
                    oneLineDown();
                    break;
            }
        }
    }

    private class GameCycle implements ActionListener {

        @Override
        public void actionPerformed(ActionEvent e) {

            doGameCycle();
        }
    }
}
```

The main logic is located in `Board`.

```
private final int BOARD_WIDTH = 10;
private final int BOARD_HEIGHT = 22;
private final int PERIOD_INTERVAL = 300;
```

We have three constants. The `BOARD_WIDTH` and `BOARD_HEIGHT` define the size of the board. The `PERIOD` defines the speed of the game.

```
...
private boolean isFallingFinished = false;
```

```
private boolean isPaused = false;
private int numLinesRemoved = 0;
private int curX = 0;
private int curY = 0;
...
```

Some important variables are initialized. The `isFallingFinished` determines if the Tetris shape has finished falling and we then need to create a new shape. The `isPaused` is used to check if the game is paused. The `numLinesRemoved` counts the number of lines that we have removed so far. The `curX` and `curY` determine the actual position of the falling Tetris shape.

```
private void initBoard(Tetris parent) {

    setPreferredSize(new Dimension(200, 400));
    setFocusable(true);

    statusbar = parent.getStatusBar();
    addKeyListener(new TAdapter());
}
```

In the `initBoard()` method, we set the preferred size of the board with the `setPreferredSize()` method. The preferred size is taken into account by the frame's `pack()` method. The focus is set with the `setFocusable()` method. We retrieve the reference to the statusbar and add the key listener with the `addKeyListener()` method.

```
private int squareWidth() {
    return (int) getSize().getWidth() / BOARD_WIDTH;
}

private int squareHeight() {
    return (int) getSize().getHeight() / BOARD_HEIGHT;
}
```

These lines determine the width and height of a single Tetrominoe square.

```
private Tetrominoe shapeAt(int x, int y) {

    return board[(y * BOARD_WIDTH) + x];
}
```

We determine the shape at the given coordinates. The shapes are stored in the `board` array.

```
void start() {

    curPiece = new Shape();
    board = new Tetrominoe[BOARD_WIDTH * BOARD_HEIGHT];

    clearBoard();
    newPiece();

    timer = new Timer(PERIOD, new GameCycle());
    timer.start();
}
```

The game is started with `start()`. We clear the board and create a new

falling piece. A timer is created. Each `PERIOD` milliseconds the `GameCycle's actionPerformed()` is called.

```
private void pause() {

    isPaused = !isPaused;

    if (isPaused) {

        statusbar.setText("paused");
    } else {

        statusbar.setText(String.valueOf(numLinesRemoved));
    }

    repaint();
}
```

The `pause()` method pauses or resumes the game. When the game is paused, we display the "paused" message in the statusbar.

Inside the `doDrawing()` method, we draw all objects on the board. The painting has two steps.

```
for (int i = 0; i < BOARD_HEIGHT; i++) {

    for (int j = 0; j < BOARD_WIDTH; j++) {

        Tetrominoe shape = shapeAt(j, BOARD_HEIGHT - i - 1);

        if (shape != Tetrominoe.NoShape) {

            drawSquare(g, j * squareWidth(),
                    boardTop + i * squareHeight(), shape);
        }
    }
}
```

In the first step we paint all the shapes or remains of the shapes that have been dropped to the bottom of the board. All the squares are remembered in the board array. We access it using the `shapeAt()` method.

```
if (curPiece.getShape() != Tetrominoe.NoShape) {

    for (int i = 0; i < 4; i++) {

        int x = curX + curPiece.x(i);
        int y = curY - curPiece.y(i);

        drawSquare(g, x * squareWidth(),
                boardTop + (BOARD_HEIGHT - y - 1)
                * squareHeight(), curPiece.getShape());
    }
}
```

In the second step, we paint the actual falling piece.

```
private void dropDown() {

    int newY = curY;
```

```
    while (newY > 0) {

        if (!tryMove(curPiece, curX, newY - 1)) {

            break;
        }

        newY--;
    }

    pieceDropped();
}
```

If we press the `Space` key, the piece is dropped to the bottom. We simply try to drop the piece one line down until it reaches the bottom or the top of another fallen Tetris piece. When the Tetris piece finishes falling, the `pieceDropped()` is called.

```
private void oneLineDown() {

    if (!tryMove(curPiece, curX, curY - 1)) {

        pieceDropped();
    }
}
```

In the `oneLineDown()` method we try to move the falling piece down one line until it is fully dropped.

```
private void clearBoard() {

    for (int i = 0; i < BOARD_HEIGHT * BOARD_WIDTH; i++) {

        board[i] = Tetrominoe.NoShape;
    }
}
```

The `clearBoard()` method fills the board with empty `Tetrominoe.NoShape`. This is later used at collision detection.

```
private void pieceDropped() {

    for (int i = 0; i < 4; i++) {

        int x = curX + curPiece.x(i);
        int y = curY - curPiece.y(i);
        board[(y * BOARD_WIDTH) + x] = curPiece.getShape();
    }

    removeFullLines();

    if (!isFallingFinished) {

        newPiece();
    }
}
```

The `pieceDropped()` method puts the falling piece into the `board` array. Once

again, the board holds all the squares of the pieces and remains of the pieces that has finished falling. When the piece has finished falling, it is time to check if we can remove some lines off the board. This is the job of the `removeFullLines()` method. Then we create a new piece, or more precisely, we try to create a new piece.

```
private void newPiece() {

    curPiece.setRandomShape();
    curX = BOARD_WIDTH / 2 + 1;
    curY = BOARD_HEIGHT - 1 + curPiece.minY();

    if (!tryMove(curPiece, curX, curY)) {

        curPiece.setShape(Tetrominoe.NoShape);
        timer.stop();

        var msg = String.format("Game over. Score: %d",
                numLinesRemoved);
        statusbar.setText(msg);
    }
}
```

The `newPiece()` method creates a new Tetris piece. The piece gets a new random shape. Then we compute the initial `curX` and `curY` values. If we cannot move to the initial positions, the game is over—we top out. The timer is stopped and we display "Game over" string on the statusbar.

```
private boolean tryMove(Shape newPiece, int newX, int newY) {

    for (int i = 0; i < 4; i++) {

        int x = newX + newPiece.x(i);
        int y = newY - newPiece.y(i);

        if (x < 0 || x >= BOARD_WIDTH || y < 0
                || y >= BOARD_HEIGHT) {

            return false;
        }

        if (shapeAt(x, y) != Tetrominoe.NoShape) {

            return false;
        }
    }

    curPiece = newPiece;
    curX = newX;
    curY = newY;

    repaint();

    return true;
}
```

The `tryMove()` method tries to move the Tetris piece. The method returns `false` if it has reached the board boundaries or it is adjacent to the already fallen Tetris pieces.

```
private void removeFullLines() {

    int numFullLines = 0;

    for (int i = BOARD_HEIGHT - 1; i >= 0; i--) {

        boolean lineIsFull = true;

        for (int j = 0; j < BOARD_WIDTH; j++) {

            if (shapeAt(j, i) == Tetrominoe.NoShape) {

                lineIsFull = false;
                break;
            }
        }

        if (lineIsFull) {

            numFullLines++;

            for (int k = i; k < BOARD_HEIGHT - 1; k++) {
                for (int j = 0; j < BOARD_WIDTH; j++) {
                    board[(k * BOARD_WIDTH) + j] =
                        shapeAt(j, k + 1);
                }
            }
        }
    }

    if (numFullLines > 0) {

        numLinesRemoved += numFullLines;

        statusbar.setText(String.valueOf(numLinesRemoved));
        isFallingFinished = true;
        curPiece.setShape(Tetrominoe.NoShape);
    }
}
```

Inside the `removeFullLines()` method we check if there is any full row among all rows in the board. If there is at least one full line, it is removed. After finding a full line we increase the counter. We move all the lines above the full row one line down. This way we destroy the full line. Notice, that in our Tetris game, we use so called naive gravity. This means that the squares may be left floating above empty gaps.

```
private void drawSquare(Graphics g, int x, int y, Tetrominoe shape) {

    Color colors[] = {new Color(0, 0, 0), new Color(204, 102, 102),
            new Color(102, 204, 102), new Color(102, 102, 204),
            new Color(204, 204, 102), new Color(204, 102, 204),
            new Color(102, 204, 204), new Color(218, 170, 0)
    };

    var color = colors[shape.ordinal()];

    g.setColor(color);
    g.fillRect(x + 1, y + 1, squareWidth() - 2, squareHeight() - 2);

    g.setColor(color.brighter());
```

```
    g.drawLine(x, y + squareHeight() - 1, x, y);
    g.drawLine(x, y, x + squareWidth() - 1, y);

    g.setColor(color.darker());
    g.drawLine(x + 1, y + squareHeight() - 1,
            x + squareWidth() - 1, y + squareHeight() - 1);
    g.drawLine(x + squareWidth() - 1, y + squareHeight() - 1,
            x + squareWidth() - 1, y + 1);
}
```

Every Tetris piece has four squares. Each of the squares is drawn with the `drawSquare()` method. Tetris pieces have different colours. The left and top sides of a square are drawn with a brighter color. Similarly, the bottom and right sides are drawn with darker colours. This is to simulate a 3D edge.

```
private void doGameCycle() {

    update();
    repaint();
}
```

The game is divided into game cycles. The `doGameCycle()` is called from the scheduled timer task. Each cycle we udpate the game and redraw the board.

```
private void update() {

    if (isPaused) {
        return;
    }

    if (isFallingFinished) {

        isFallingFinished = false;
        newPiece();
    } else {

        oneLineDown();
    }
}
```

The `update()` represents one step of the game. The falling piece goes one line down or a new piece is created if the previous one has finished falling.

```
private class TAdapter extends KeyAdapter {

    @Override
    public void keyPressed(KeyEvent e) {
        ...
```

The game is controlled with cursor keys. We check for key events in the `KeyAdapter`.

```
int keycode = e.getKeyCode();
```

We get the key code with the `getKeyCode()` method.

```
if (keycode == KeyEvent.VK_P) {
    pause();
    return;
```

```
}
```

If we press the `P` key, the game is paused.

```
switch (keycode) {

    case KeyEvent.VK_LEFT:
        tryMove(curPiece, curX - 1, curY);
        break;
    ...
```

When we press the left cursor key, we try to move the falling piece to the left.

```
case KeyEvent.VK_DOWN:
    tryMove(curPiece.rotateRight(), curX, curY);
    break;
```

With the down cursor key, the piece is rotated to the right if possible.

```
case KeyEvent.VK_UP:
    tryMove(curPiece.rotateLeft(), curX, curY);
    break;
```

With the up cursor key, the piece is rotated to the left if possible.

```
private class GameCycle implements ActionListener {

    @Override
    public void actionPerformed(ActionEvent e) {

        doGameCycle();
    }
}
```

The timer periodically calls the `GameCycle's actionPerformed()` method. Inside the method, we call the `doGameCycle()` method.

Listing 3.3: Tetris.java

```
package com.zetcode;

import java.awt.BorderLayout;
import java.awt.EventQueue;
import javax.swing.JFrame;
import javax.swing.JLabel;

/**
 * Java Tetris game
 *
 * Author: Jan Bodnar
 * Website: http://zetcode.com
 */

public class Tetris extends JFrame {

    private JLabel statusbar;

    public Tetris() {

        initUI();
```

```
    }

    private void initUI() {

        statusbar = new JLabel(" 0");
        add(statusbar, BorderLayout.SOUTH);

        var board = new Board(this);
        add(board);
        board.start();

        setResizable(false);
        pack();

        setTitle("Tetris");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
    }

    public JLabel getStatusBar() {

        return statusbar;
    }

    public static void main(String[] args) {

        EventQueue.invokeLater(() -> {

            var game = new Tetris();
            game.setVisible(true);
        });
    }
}
```

In `Tetris` we set up the Tetris game.

```
statusbar = new JLabel(" 0");
add(statusbar, BorderLayout.SOUTH);
```

The score is displayed in a label which is located at the bottom of the board.

```
var board = new Board(this);
add(board);
board.start();
```

The board is created and added to the container. The `start()` method starts the game.

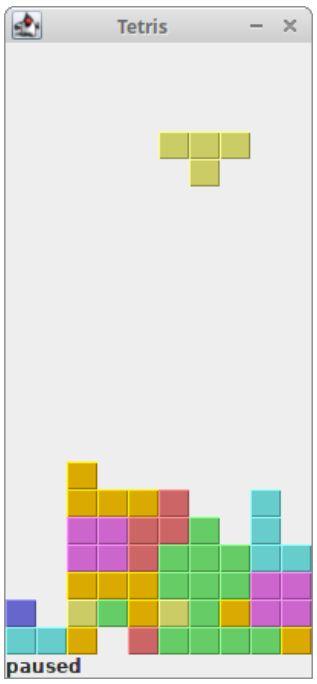

Figure 3.3: Tetris

Chapter 4

Minesweeper

Minesweeper is a popular board game shipped with many operating systems by default. The goal of the game is to sweep all mines from a mine field. If the player clicks on the cell which contains a mine, the mine detonates and the game is over.

A cell can contain a number or it can be blank. The number indicates how many mines are adjacent to this particular cell. We set a mark on a cell by right clicking on it. This way we indicate that we believe that there is a mine.

Listing 4.1: Board.java

```
package com.zetcode;

import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.util.Random;
import javax.swing.ImageIcon;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class Board extends JPanel {

    private final int NUM_IMAGES = 13;
    private final int CELL_SIZE = 15;

    private final int COVER_FOR_CELL = 10;
    private final int MARK_FOR_CELL = 10;
    private final int EMPTY_CELL = 0;
    private final int MINE_CELL = 9;
    private final int COVERED_MINE_CELL = MINE_CELL
            + COVER_FOR_CELL;
    private final int MARKED_MINE_CELL = COVERED_MINE_CELL
            + MARK_FOR_CELL;

    private final int DRAW_MINE = 9;
    private final int DRAW_COVER = 10;
    private final int DRAW_MARK = 11;
    private final int DRAW_WRONG_MARK = 12;
```

```
    private final int N_MINES = 40;
    private final int N_ROWS = 16;
    private final int N_COLS = 16;

    private final int BOARD_WIDTH = N_ROWS * CELL_SIZE + 1;
    private final int BOARD_HEIGHT = N_COLS * CELL_SIZE + 1;

    private int[] field;
    private boolean inGame;
    private int minesLeft;
    private Image[] img;

    private int allCells;
    private final JLabel statusbar;

    public Board(JLabel statusbar) {

        this.statusbar = statusbar;
        initBoard();
    }

    private void initBoard() {

        setPreferredSize(new Dimension(BOARD_WIDTH, BOARD_HEIGHT));

        img = new Image[NUM_IMAGES];

        for (int i = 0; i < NUM_IMAGES; i++) {

            var path = "src/resources/" + i + ".png";
            img[i] = (new ImageIcon(path)).getImage();
        }

        addMouseListener(new MinesAdapter());
        newGame();
    }

    private void newGame() {

        int cell;

        var random = new Random();
        inGame = true;
        minesLeft = N_MINES;

        allCells = N_ROWS * N_COLS;
        field = new int[allCells];

        for (int i = 0; i < allCells; i++) {

            field[i] = COVER_FOR_CELL;
        }

        statusbar.setText(Integer.toString(minesLeft));

        int i = 0;

        while (i < N_MINES) {

            int position = (int) (allCells * random.nextDouble());

            if ((position < allCells)
```

```
    && (field[position] != COVERED_MINE_CELL)) {

int current_col = position % N_COLS;
field[position] = COVERED_MINE_CELL;
i++;

if (current_col > 0) {

    cell = position - 1 - N_COLS;

    if (cell >= 0) {

        if (field[cell] != COVERED_MINE_CELL) {
            field[cell] += 1;
        }
    }

    cell = position - 1;

    if (cell >= 0) {

        if (field[cell] != COVERED_MINE_CELL) {

            field[cell] += 1;
        }
    }

    cell = position + N_COLS - 1;

    if (cell < allCells) {

        if (field[cell] != COVERED_MINE_CELL) {

            field[cell] += 1;
        }
    }
}

cell = position - N_COLS;

if (cell >= 0) {

    if (field[cell] != COVERED_MINE_CELL) {

        field[cell] += 1;
    }
}

cell = position + N_COLS;

if (cell < allCells) {

    if (field[cell] != COVERED_MINE_CELL) {

        field[cell] += 1;
    }
}

if (current_col < (N_COLS - 1)) {

    cell = position - N_COLS + 1;
```

```
                if (cell >= 0) {

                    if (field[cell] != COVERED_MINE_CELL) {

                        field[cell] += 1;
                    }
                }

                cell = position + N_COLS + 1;

                if (cell < allCells) {

                    if (field[cell] != COVERED_MINE_CELL) {

                        field[cell] += 1;
                    }
                }

                cell = position + 1;

                if (cell < allCells) {

                    if (field[cell] != COVERED_MINE_CELL) {

                        field[cell] += 1;
                    }
                }
            }
        }
    }
}

public void find_empty_cells(int j) {

    int current_col = j % N_COLS;
    int cell;

    if (current_col > 0) {

        cell = j - N_COLS - 1;

        if (cell >= 0) {

            if (field[cell] > MINE_CELL) {

                field[cell] -= COVER_FOR_CELL;

                if (field[cell] == EMPTY_CELL) {

                    find_empty_cells(cell);
                }
            }
        }

        cell = j - 1;

        if (cell >= 0) {

            if (field[cell] > MINE_CELL) {

                field[cell] -= COVER_FOR_CELL;
```

```
                if (field[cell] == EMPTY_CELL) {

                    find_empty_cells(cell);
                }
            }
        }

        cell = j + N_COLS - 1;

        if (cell < allCells) {

            if (field[cell] > MINE_CELL) {

                field[cell] -= COVER_FOR_CELL;

                if (field[cell] == EMPTY_CELL) {

                    find_empty_cells(cell);
                }
            }
        }
    }

    cell = j - N_COLS;

    if (cell >= 0) {

        if (field[cell] > MINE_CELL) {

            field[cell] -= COVER_FOR_CELL;

            if (field[cell] == EMPTY_CELL) {

                find_empty_cells(cell);
            }
        }
    }

    cell = j + N_COLS;

    if (cell < allCells) {

        if (field[cell] > MINE_CELL) {

            field[cell] -= COVER_FOR_CELL;

            if (field[cell] == EMPTY_CELL) {

                find_empty_cells(cell);
            }
        }
    }

    if (current_col < (N_COLS - 1)) {

        cell = j - N_COLS + 1;

        if (cell >= 0) {

            if (field[cell] > MINE_CELL) {

                field[cell] -= COVER_FOR_CELL;
```

```
                if (field[cell] == EMPTY_CELL) {

                    find_empty_cells(cell);
                }
            }
        }

        cell = j + N_COLS + 1;

        if (cell < allCells) {

            if (field[cell] > MINE_CELL) {

                field[cell] -= COVER_FOR_CELL;

                if (field[cell] == EMPTY_CELL) {

                    find_empty_cells(cell);
                }
            }
        }

        cell = j + 1;

        if (cell < allCells) {

            if (field[cell] > MINE_CELL) {

                field[cell] -= COVER_FOR_CELL;

                if (field[cell] == EMPTY_CELL) {

                    find_empty_cells(cell);
                }
            }
        }
    }
}

@Override
public void paintComponent(Graphics g) {

    int uncover = 0;

    for (int i = 0; i < N_ROWS; i++) {

        for (int j = 0; j < N_COLS; j++) {

            int cell = field[(i * N_COLS) + j];

            if (inGame && cell == MINE_CELL) {

                inGame = false;
            }

            if (!inGame) {

                if (cell == COVERED_MINE_CELL) {

                    cell = DRAW_MINE;
                } else if (cell == MARKED_MINE_CELL) {
```

```
                    cell = DRAW_MARK;
                } else if (cell > COVERED_MINE_CELL) {

                    cell = DRAW_WRONG_MARK;
                } else if (cell > MINE_CELL) {

                    cell = DRAW_COVER;
                }

            } else {

                if (cell > COVERED_MINE_CELL) {

                    cell = DRAW_MARK;
                } else if (cell > MINE_CELL) {

                    cell = DRAW_COVER;
                    uncover++;
                }
            }

            g.drawImage(img[cell], (j * CELL_SIZE),
                    (i * CELL_SIZE), this);
        }
    }

    if (uncover == 0 && inGame) {

        inGame = false;
        statusbar.setText("Game won");

    } else if (!inGame) {

        statusbar.setText("Game lost");
    }
}

private class MinesAdapter extends MouseAdapter {

    @Override
    public void mousePressed(MouseEvent e) {

        int x = e.getX();
        int y = e.getY();

        int cCol = x / CELL_SIZE;
        int cRow = y / CELL_SIZE;

        boolean doRepaint = false;

        if (!inGame) {

            newGame();
            repaint();
        }

        if ((x < N_COLS * CELL_SIZE) && (y < N_ROWS * CELL_SIZE)) {

            if (e.getButton() == MouseEvent.BUTTON3) {

                if (field[(cRow * N_COLS) + cCol] > MINE_CELL) {
```

```
                doRepaint = true;

                if (field[(cRow * N_COLS) + cCol]
                    <= COVERED_MINE_CELL) {

                    if (minesLeft > 0) {

                        field[(cRow * N_COLS) + cCol]
                                        += MARK_FOR_CELL;
                        minesLeft--;
                        var msg = Integer.toString(minesLeft);
                        statusbar.setText(msg);

                    } else {
                        statusbar.setText("No marks left");
                    }
                } else {

                    field[(cRow * N_COLS) + cCol]
                                        -= MARK_FOR_CELL;
                    minesLeft++;
                    var msg = Integer.toString(minesLeft);
                    statusbar.setText(msg);
                }
            }

        } else {

            if (field[(cRow * N_COLS) + cCol]
                            > COVERED_MINE_CELL) {

                return;
            }

            if ((field[(cRow * N_COLS) + cCol] > MINE_CELL)
                    && (field[(cRow * N_COLS) + cCol]
                    < MARKED_MINE_CELL)) {

                field[(cRow * N_COLS) + cCol]
                        -= COVER_FOR_CELL;
                doRepaint = true;

                if (field[(cRow * N_COLS) + cCol] == MINE_CELL)
                {

                    inGame = false;
                }

                if (field[(cRow * N_COLS)+cCol] == EMPTY_CELL)
                {

                    find_empty_cells((cRow * N_COLS) + cCol);
                }
            }
        }

        if (doRepaint) {

            repaint();
        }
    }
```

```
        }
    }
}
```

We begin with defining constants in the game.

```
private final int NUM_IMAGES = 13;
private final int CELL_SIZE = 15;
```

There are thirteen images used in this game. A cell can be surrounded by maximum of eight mines, so we need numbers one through eight. We need images for an empty cell, a mine, a covered cell, a marked cell and finally for a wrongly marked cell. The size of each of the images is 15x15 px.

```
private final int COVER_FOR_CELL = 10;
private final int MARK_FOR_CELL = 10;
private final int EMPTY_CELL = 0;
```

A mine field is an array of numbers. For example, 0 denotes an empty cell. Number 10 is used for a cell cover as well as for a mark. Using constants improves readability of the code.

```
private final int MINE_CELL = 9;
```

The `MINE_CELL` represents a cell that contains a mine.

```
private final int COVERED_MINE_CELL = MINE_CELL
        + COVER_FOR_CELL;
private final int MARKED_MINE_CELL = COVERED_MINE_CELL
        + MARK_FOR_CELL;
```

The `COVERED_MINE_CELL` is used for a field that is covered and contains a mine. The `MARKED_MINE_CELL` is a covered mine cell that was marked by the user.

```
private final int DRAW_MINE = 9;
private final int DRAW_COVER = 10;
private final int DRAW_MARK = 11;
private final int DRAW_WRONG_MARK = 12;
```

These contants determine whether to draw a mine, a mine cover, a mark, and a wrongly marked cell.

```
private final int N_MINES = 40;
private final int N_ROWS = 16;
private final int N_COLS = 16;
```

The minefield in our game has fourty hidden mines. There are sixteen rows and sixteen columns in the field. So there are two hundred and twenty-six cells together in the minefield.

```
private int[] field;
```

The field is an array of numbers. Each cell in the field has a specific number. For instance, a mine cell has number 9. A cell with number 2 means it is adjacent to two mines. The numbers are added. For example, a covered mine has number 19, 9 for the mine and 10 for the cell cover and so on.

```
private boolean inGame;
```

The `inGame` variable determines whether we are in the game or the game is over.

```
private int minesLeft;
```

The `minesLeft` variable the number of mines to be marked left.

```
for (int i = 0; i < NUM_IMAGES; i++) {

    var path = "src/resources/" + i + ".png";
    img[i] = (new ImageIcon(path)).getImage();
}
```

We load our images into the image array. The images are named 0.png, 1.png ... 12.png.

```
allCells = N_ROWS * N_COLS;
field = new int[allCells];

for (int i = 0; i < allCells; i++) {

    field[i] = COVER_FOR_CELL;
}
```

The `newGame()` initiates the Minesweeper game. These lines set up the mine field. Every cell is covered by default.

```
int i = 0;

while (i < N_MINES) {

    int position = (int) (allCells * random.nextDouble());

    if ((position < allCells)
            && (field[position] != COVERED_MINE_CELL)) {

        int current_col = position % N_COLS;
        field[position] = COVERED_MINE_CELL;
        i++;
...
```

In the while cycle we randomly position all mines in the field.

```
cell = position - N_COLS;

if (cell >= 0) {

    if (field[cell] != COVERED_MINE_CELL) {

        field[cell] += 1;
    }
}
```

Each of the cells can be surrounded up to eight cells. (This does not apply to the border cells.) We raise the number for adjacent cells for each of the randomly placed mine. In our example, we add 1 to the top neighbour of the cell in question.

In the `find_empty_cells()` method, we find empty cells. If the player clicks on a mine cell, the game is over. If he clicks on a cell adjacent to a mine, he uncovers a number indicating how many mines the cell is adjacent to. Clicking on an empty cell leads to uncovering many other empty cells plus cells with a number that form a border around a space of empty borders. We use a recursive algorithm to find empty cells.

```
cell = j - 1;

if (cell >= 0) {

    if (field[cell] > MINE_CELL) {

        field[cell] -= COVER_FOR_CELL;

        if (field[cell] == EMPTY_CELL) {

            find_empty_cells(cell);
        }
    }
}
```

In this code, we check the cell that is located to the left to an empty cell in question. If it is not empty, it is uncovered. If it is empty, we repeat the whole process by recursively calling the `find_empty_cells()` method.

The `paintComponent()` method turns numbers into images.

```
if (!inGame) {

    if (cell == COVERED_MINE_CELL) {

        cell = DRAW_MINE;
    } else if (cell == MARKED_MINE_CELL) {

        cell = DRAW_MARK;
    } else if (cell > COVERED_MINE_CELL) {

        cell = DRAW_WRONG_MARK;
    } else if (cell > MINE_CELL) {

        cell = DRAW_COVER;
    }
} ...
```

If the game is over and we lost, we show all uncovered mines if any and show all wrongly marked cells if any.

```
g.drawImage(img[cell], (j * CELL_SIZE),
    (i * CELL_SIZE), this);
```

This code line draws every cell on the window. A cell is drawn with the `drawImage()` method.

```
if (uncover == 0 && inGame) {

    inGame = false;
    statusbar.setText("Game won");
```

```
} else if (!inGame) {

    statusbar.setText("Game lost");
}
```

If there is nothing left to uncover, we win. If the `inGame` variable was set to `false`, we have lost.

In the `mousePressed()` method we react to mouse clicks. The Minesweeper game is controlled solely by mouse. We react to left and right mouse clicks.

```
int x = e.getX();
int y = e.getY();
```

We determine the `x` and `y` coordinates of the mouse pointer.

```
int cCol = x / CELL_SIZE;
int cRow = y / CELL_SIZE;
```

We compute the corresponding column and row of the mine field.

```
if ((x < N_COLS * CELL_SIZE) && (y < N_ROWS * CELL_SIZE)) {
```

We check that we are located in the area of the mine field.

```
if (e.getButton() == MouseEvent.BUTTON3) {
```

The uncovering of the mines is done with the right mouse button.

```
field[(cRow * N_COLS) + cCol] += MARK_FOR_CELL;
minesLeft--;
```

If we right click on an unmarked cell, we add `MARK_FOR_CELL` to the number representing the cell. This leads to drawing a covered cell with a mark in the `paintComponent()` method.

```
field[(cRow * N_COLS) + cCol] -= MARK_FOR_CELL;
minesLeft++;
var msg = Integer.toString(minesLeft);
statusbar.setText(msg);
```

If we left click on a cell that has been already marked, we remove the mark and increase the number of cells to be marked.

```
if (field[(cRow * N_COLS) + cCol] > COVERED_MINE_CELL) {

    return;
}
```

Nothing happens if we click on the covered and marked cell. It must by first uncovered by another right click and only then it is possible to left click on it.

```
field[(cRow * N_COLS) + cCol] -= COVER_FOR_CELL;
```

A left click removes a cover from the cell.

```
if (field[(cRow * N_COLS) + cCol] == MINE_CELL) {
```

```
    inGame = false;
}

if (field[(cRow * N_COLS) + cCol] == EMPTY_CELL) {

    find_empty_cells((cRow * N_COLS) + cCol);
}
```

In case we left clicked on a mine, the game is over. If we left clicked on an empty cell, we call the `find_empty_cells()` method which recursively finds all adjacent empty cells.

```
if (doRepaint) {

    repaint();
}
```

If the board needs to be repainted (for instance a mark was set or removed), we call the `repaint()` method.

Listing 4.2: Minesweeper.java

```
package com.zetcode;

import java.awt.BorderLayout;
import java.awt.EventQueue;
import javax.swing.JFrame;
import javax.swing.JLabel;

/**
 * Java Minesweeper game
 *
 * Author: Jan Bodnar
 * Website: http://zetcode.com
 */

public class Minesweeper extends JFrame {

    private JLabel statusbar;

    public Minesweeper() {

        initUI();
    }

    private void initUI() {

        statusbar = new JLabel("");
        add(statusbar, BorderLayout.SOUTH);

        add(new Board(statusbar));

        setResizable(false);
        pack();

        setTitle("Minesweeper");
        setLocationRelativeTo(null);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
```

```
    public static void main(String[] args) {

        EventQueue.invokeLater(() -> {

            var game = new Minesweeper();
            game.setVisible(true);
        });
    }
}
```

This is the main class.

```
setResizable(false);
```

The window has fixed size. For this, we use the `setResizable()` method.

Figure 4.1: Minesweeper

Chapter 5

Balloons

Balloons is a simple 2D game in which the player cracks flying balloons. The game is controlled with the mouse. The player moves the aiming target and tries to crack all balloons. He misses if the balloon passes the top of the board. There are several types of balloons; they have different colour and speed.

A sprite is a two-dimensional bitmap that is integrated into a larger scene. In the Balloons game, we have two sprites: the balloon and the target.

Listing 5.1: Sprite.java

```
package com.zetcode;

import java.awt.Image;
import javax.swing.ImageIcon;

public class Sprite {

    protected double x;
    protected double y;
    protected int width;
    protected int height;
    protected boolean visible;
    protected Image image;

    public Sprite(double x, double y) {

        this.x = x;
        this.y = y;

        visible = true;
    }

    protected void loadImage(String imageName) {

        var ii = new ImageIcon(imageName);
        image = ii.getImage();
    }

    protected void getImageDimensions() {

        width = image.getWidth(null);
        height = image.getHeight(null);
    }
```

```
    public Image getImage() {

        return image;
    }

    public double getX() {

        return x;
    }

    public double getY() {

        return y;
    }

    public boolean isVisible() {

        return visible;
    }

    public void setVisible(Boolean visible) {

        this.visible = visible;
    }
}
```

The `Sprite` represents a sprite in the game. It contains common attributes including the `x` and `y` coordinates, `width` and `height`, `visible` state and `image`.

Listing 5.2: Balloon.java

```
package com.zetcode;

import java.util.Random;

public class Balloon extends Sprite {

    private final int NUMBER_OF_BALLOON_TYPES = 5;
    private int speed = 2;

    public Balloon(int x, int y) {
        super(x, y);

        initBalloon();
    }

    private void initBalloon() {

        int r = new Random().nextInt(NUMBER_OF_BALLOON_TYPES) + 1;
        speed = Math.max(2, r);
        loadImage(String.format("src/resources/balloon%d.png", r));

        getImageDimensions();
    }

    public void move() {

        y -= speed;
    }
```

```
}
```

The `Balloon` represents the balloon sprite. We have five different balloon types. Each of them has a different colour.

```
int r = new Random().nextInt(NUMBER_OF_BALLOON_TYPES) + 1;
speed = Math.max(2, r);
loadImage(String.format("src/resources/balloon%d.png", r));
```

The balloon types are randomly generated. There are different types of PNG images for different balloon types. Also, the speed of the balloons is randomly set.

```
public void move() {

    y -= speed;
}
```

The balloons move from the bottom to the top of the board. The coordinate system origin is in the upper-left corner of the board, where the `x` coordinate values increase to the right, and `y` coordinate values increase to the bottom.

Listing 5.3: Target.java

```
package com.zetcode;

import java.awt.event.MouseEvent;

public class Target extends Sprite {

    public Target(double x, double y) {
        super(x, y);

        initTarget();
    }

    private void initTarget() {

        loadImage("src/resources/target.png");
        getImageDimensions();
    }


    public void mouseMoved(MouseEvent e) {

        x = e.getX();
        y = e.getY();
    }
}
```

`Target` represents the aiming target.

```
private void initTarget() {

    loadImage("src/resources/target.png");
    getImageDimensions();
}
```

In the `initTarget()` we load the image for the sprite and get its width and height.

```
public void mouseMoved(MouseEvent e) {

    x = e.getX();
    y = e.getY();
}
```

When the mouse is moved, the sprite's `x` and `y` coordinates are updated. The coordinates are retrieved from the `MouseEvent`.

Listing 5.4: Board.java

```
package com.zetcode;

import javax.swing.JPanel;
import java.awt.Color;
import java.awt.Cursor;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.MouseInfo;
import java.awt.Point;
import java.awt.RenderingHints;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionAdapter;
import java.awt.geom.Ellipse2D;
import java.awt.image.BufferedImage;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import javax.swing.Timer;

/**
 * Java Balloons game
 *
 * Author: Jan Bodnar
 * Website: http://zetcode.com
 */

public class Board extends JPanel {

    protected static final int BOARD_WIDTH = 600;
    protected static final int BOARD_HEIGHT = 400;

    private final int NUM_OF_BALLOONS = 20;
    private final int BALLOON_WIDTH = 40;
    private final int BALLOON_HEIGHT = 60;
    private final int TARGET_WIDTH = 24;

    private List<Balloon> balloons;

    private final int FPS = 60;
    private final int PERIOD = 1000 / FPS;
```

```
    private int balloonsCracked = 0;
    private int balloonsMissed = 0;
    private Target target;

    private Timer timer;

    private boolean isRunning = true;

    public Board() {

        initBoard();
    }

    private void initBoard() {

        setPreferredSize(
            new Dimension(Board.BOARD_WIDTH, Board.BOARD_HEIGHT));

        // hides cursor
        setCursor(getToolkit().createCustomCursor(
                new BufferedImage(1, 1, BufferedImage.TYPE_INT_ARGB),
                new Point(0, 0),
                null));

        createBalloons();

        addMouseMotionListener(new MAdapter());
        addMouseListener(new MAdapter2());
        setBackground(Color.GRAY);

        Point p = MouseInfo.getPointerInfo().getLocation();
        target = new Target(p.getX(), p.getY());

        timer = new Timer(PERIOD, new GameCycle());
        timer.start();
    }

    private void createBalloons() {

        balloons = new ArrayList<>();
        int startY = 1200;

        for (int i = 0; i < NUM_OF_BALLOONS; i++) {

            int x = new Random().nextInt(BOARD_WIDTH - BALLOON_WIDTH);
            int y = new Random().nextInt(startY) + BOARD_HEIGHT;

            var balloon = new Balloon(x, y);
            balloons.add(balloon);
        }
    }

    private void doGameCycle() {

        updateBalloons();
        repaint();
    }

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

```
        if (isRunning) {

            doDrawing(g);
        } else {

            gameOver(g);
        }

        Toolkit.getDefaultToolkit().sync();
    }

    private void doDrawing(Graphics g) {

        var g2d = (Graphics2D) g;

        drawScore(g2d);

        for (Balloon balloon : balloons) {

            g2d.drawImage(balloon.getImage(), (int) balloon.getX(),
                    (int) balloon.getY(), this);
        }

        g2d.drawImage(target.getImage(), (int) target.getX(),
                (int) target.getY(), this);
    }

    private void updateBalloons() {

        for (int i = 0; i < balloons.size(); i++) {

            var balloon = balloons.get(i);

            double by = balloon.getY();

            if (by + BALLOON_HEIGHT < 0) {

                balloonsMissed++;
                balloon.setVisible(false);
            }

            if (balloon.isVisible()) {

                balloon.move();
            } else {

                balloons.remove(i);

                if (balloons.isEmpty()) {

                    timer.stop();
                    isRunning = false;
                    setCursor(Cursor.getDefaultCursor());
                }
            }
        }
    }

    private void fire(int mx, int my) {

        for (int i = 0; i < balloons.size(); i++) {
```

```
            var balloon = balloons.get(i);

            double bx = balloon.getX();
            double by = balloon.getY();

            var ellipse = new Ellipse2D.Double(bx, by,
                    BALLOON_WIDTH, BALLOON_HEIGHT);

            if (ellipse.contains(mx, my)) {

                balloon.setVisible(false);
                balloonsCracked++;
            }
        }
    }

    private void drawScore(Graphics2D g2d) {

        g2d.setFont(new Font("Geneva", Font.BOLD, 12));

        g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                RenderingHints.VALUE_ANTIALIAS_ON);

        g2d.setRenderingHint(RenderingHints.KEY_RENDERING,
                RenderingHints.VALUE_RENDER_QUALITY);

        var label1 = String.format("Destroyed %d", balloonsCracked);
        g2d.drawString(label1, 5, BOARD_HEIGHT - 85);

        int nOfBalloons = balloons.size();
        var label2 = String.format("Left %d", nOfBalloons);
        g2d.drawString(label2, 5, BOARD_HEIGHT - 60);

        var label3 = String.format("Missed %d", balloonsMissed);
        g2d.drawString(label3, 5, BOARD_HEIGHT - 35);
    }

    private void gameOver(Graphics g) {

        var g2d = (Graphics2D) g;

        g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                RenderingHints.VALUE_ANTIALIAS_ON);

        g2d.setRenderingHint(RenderingHints.KEY_RENDERING,
                RenderingHints.VALUE_RENDER_QUALITY);

        var msg = "Game Over";
        var msg2 = String.format("Cracked: %d missed: %d",
                balloonsCracked, balloonsMissed);

        var myFont = new Font("Geneva", Font.BOLD, 24);
        var fontMetrics = this.getFontMetrics(myFont);

        g.setFont(myFont);
        g.drawString(msg,
                (BOARD_WIDTH - fontMetrics.stringWidth(msg)) / 2,
                (BOARD_HEIGHT / 2) - fontMetrics.getHeight());

        g.drawString(msg2, (BOARD_WIDTH -
                fontMetrics.stringWidth(msg2)) / 2,
                (BOARD_HEIGHT / 2) + fontMetrics.getHeight());
```

```
    }

    private class MAdapter extends MouseMotionAdapter {

        @Override
        public void mouseMoved(MouseEvent e) {
            target.mouseMoved(e);
        }
    }

    private class MAdapter2 extends MouseAdapter {

        @Override
        public void mousePressed(MouseEvent e) {

            int mx = e.getX() + TARGET_WIDTH / 2;
            int my = e.getY() + TARGET_WIDTH / 2;

            fire(mx, my);
        }
    }

    private class GameCycle implements ActionListener {

        @Override
        public void actionPerformed(ActionEvent e) {

            doGameCycle();
        }
    }
}
```

The game logic is in the `Board`.

```
protected static final int BOARD_WIDTH = 600;
protected static final int BOARD_HEIGHT = 400;
```

The `BOARD_WIDTH` and `BOARD_HEIGHT` constants store the width and height of the game window.

```
private final int NUM_OF_BALLOONS = 20;
private final int BALLOON_WIDTH = 40;
private final int BALLOON_HEIGHT = 60;
private final int TARGET_WIDTH = 24;
```

The number of balloons is stored in the `NUM_OF_BALLOONS`. The `BALLOON_WIDTH` stores the width of the balloon sprite; it is needed when positioning a new balloon on the board. The `BALLOON_HEIGHT` stores the height of the balloon; it is needed when the balloons disapper from the board. The `TARGET_WIDTH` stores the width of the aiming target; it is needed when determining the exact location of the shots fired.

```
private List<Balloon> balloons;
```

All the balloons are stored in the `balloons` list.

```
private final int FPS = 60;
```

The `FPS` constant defines the number of frames per second in a game animation. The number determines the quality of the animation in a film or a game. For simple 2D single player games, FPS between 30 and 60 is good enough.

```
private final int PERIOD = 1000 / FPS;
```

The `PERIOD` defines the period between successive executions of a scheduled timer, which creates a game loop.

```
private int balloonsCracked = 0;
private int balloonsMissed = 0;
```

These two variables store the number of cracked and missed balloons.

```
private void initBoard() {
```

We have put some initialization code into the `initBoard()` method.

```
// hides cursor
setCursor(getToolkit().createCustomCursor(
        new BufferedImage(1, 1, BufferedImage.TYPE_INT_ARGB),
        new Point(0, 0),
        null));
```

We hide the mouse cursor by implementing a tiny transparent custom cursor. An aiming target is displayed instead.

```
createBalloons();

addMouseMotionListener(new MAdapter());
addMouseListener(new MAdapter2());
```

We create the balloons and add mouse listeners.

```
Point p = MouseInfo.getPointerInfo().getLocation();

target = new Target(p.getX(), p.getY());
```

We initiate the aiming target. The `PointerInfo` represents the current location of the mouse pointer.

```
timer = new Timer(PERIOD, new GameCycle());
timer.start();
```

A timer is created.

```
private void createBalloons() {

    balloons = new ArrayList<>();
    int startY = 1200;

    for (int i = 0; i < NUM_OF_BALLOONS; i++) {

        int x = new Random().nextInt(BOARD_WIDTH - BALLOON_WIDTH);
        int y = new Random().nextInt(startY) + BOARD_HEIGHT;

        var balloon = new Balloon(x, y);
        balloons.add(balloon);
```

```
    }
}
```

We create the balloons in a for loop. We place them in the `balloons` list. The `x` coordinate is randomly generated within the bounds of a board. The `y` coordinate is also randomly set so that the balloons appear on the board at different times.

```
private void doGameCycle() {

    updateBalloons();
    repaint();
}
```

Inside the `doGameCycle()`, we update the positions of the balloons and repaint the board with `repaint()`.

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

    if (isRunning) {

        doDrawing(g);
    } else {

        gameOver(g);
    }

    Toolkit.getDefaultToolkit().sync();
}
```

The `repaint()` method causes the `paintComponent()` to be launched. Depending on the `isRunning` variable, we either call the `doDrawing()` method or the `gameOver()` method.

```
private void doDrawing(Graphics g) {

    var g2d = (Graphics2D) g;

    drawScore(g2d);

    for (Balloon balloon : balloons) {

        g2d.drawImage(balloon.getImage(), (int) balloon.getX(),
                (int) balloon.getY(), this);
    }

    g2d.drawImage(target.getImage(), (int) target.getX(),
            (int) target.getY(), this);
}
```

Inside the `doDrawing()` method, we draw the score, the balloons and the target. The balloons and the aiming target are images and are drawn with `drawImage()`.

```
private void updateBalloons() {

    for (int i = 0; i < balloons.size(); i++) {
```

```
        var balloon = balloons.get(i);

        double by = balloon.getY();
...
```

The `updateBalloons()` updates the balloons. We go through the list of the balloons and get their `y` coordinates.

```
if (by + BALLOON_HEIGHT < 0) {

    balloonsMissed++;
    balloon.setVisible(false);
}
```

If the balloon passes the top of the board, we increase the `balloonsMissed` variable and hide the balloon with `setVisible()`.

```
if (balloon.isVisible()) {

    balloon.move();
} else {

    balloons.remove(i);

    if (balloons.isEmpty()) {

        timer.stop();
        isRunning = false;
        setCursor(Cursor.getDefaultCursor());
    }
}
```

The balloons that are visible are moved. Otherwise, they are removed from the list of the balloons. If the list is empty, the timer is stopped, the `isRunning` variable is set to `false` and the cursor is set back to the default one.

```
private void fire(int mx, int my) {

    for (int i = 0; i < balloons.size(); i++) {

        var balloon = balloons.get(i);

        double bx = balloon.getX();
        double by = balloon.getY();

        var ellipse = new Ellipse2D.Double(bx, by,
                BALLOON_WIDTH, BALLOON_HEIGHT);

        if (ellipse.contains(mx, my)) {

            balloon.setVisible(false);
            balloonsCracked++;
        }
    }
}
```

The `fire()` method is called when we left-click with the mouse. In the method we test whether we have clicked inside the area of any of the balloons in the list. We receive the coordinates of the shot in the arguments and determine the

coordinates of the balloons. If the mouse coordinates are inside the ellipse of the balloon (tested with `contains()`), the balloon is cracked.

```
private void drawScore(Graphics2D g2d) {
...
```

Inside the `drawScore()` we draw the number of destroyed, missed, and left balloons. The stats are located at the bottom-left corner of the window.

```
var label1 = String.format("Destroyed %d", balloonsCracked);
g2d.drawString(label1, 5, BOARD_HEIGHT - 85);
```

We format the message with `String.format()` and draw it on the board with `drawString()`.

```
private void gameOver(Graphics g) {
```

The `gameOver()` method draws the ending message on the board. It includes the number of cracked and missed balloons.

```
var myFont = new Font("Geneva", Font.BOLD, 24);
var fontMetrics = this.getFontMetrics(myFont);
```

We get the `FontMetrics` to center the message on the window.

```
g.drawString(msg,
        (BOARD_WIDTH - fontMetrics.stringWidth(msg)) / 2,
        (BOARD_HEIGHT / 2) - fontMetrics.getHeight());
```

We can get the width of the string with `stringWidth()` and the height of the text line with `getHeight()`.

```
private class MAdapter extends MouseMotionAdapter {

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

The `MouseMotionAdapter` receives mouse motion events. In the `mouseMoved()` method we update the coordinates of the aiming target.

```
private class MAdapter2 extends MouseAdapter {

    @Override
    public void mousePressed(MouseEvent e) {

        int mx = e.getX() + TARGET_WIDTH / 2;
        int my = e.getY() + TARGET_WIDTH / 2;

        fire(mx, my);
    }
}
```

The `MouseAdapter` receives mouse events. In the `mousePressed()` method we get the coordinates of the shot and call the `fire()` method.

Listing 5.5: Balloons.java

```
package com.zetcode;

import java.awt.EventQueue;
import javax.swing.JFrame;


/**
 * Java Balloons game
 *
 * Author: Jan Bodnar
 * Website: http://zetcode.com
 */

public class Balloons extends JFrame {

    public Balloons() {

        initUI();
    }

    private void initUI() {

        var board = new Board();
        add(board);

        setResizable(false);

        pack();

        setTitle("Balloons");
        setLocationRelativeTo(null);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }


    public static void main(String[] args) {

        EventQueue.invokeLater(() -> {

            var game = new Balloons();
            game.setVisible(true);
        });
    }
}
```

In `Balloons` we set up the game.

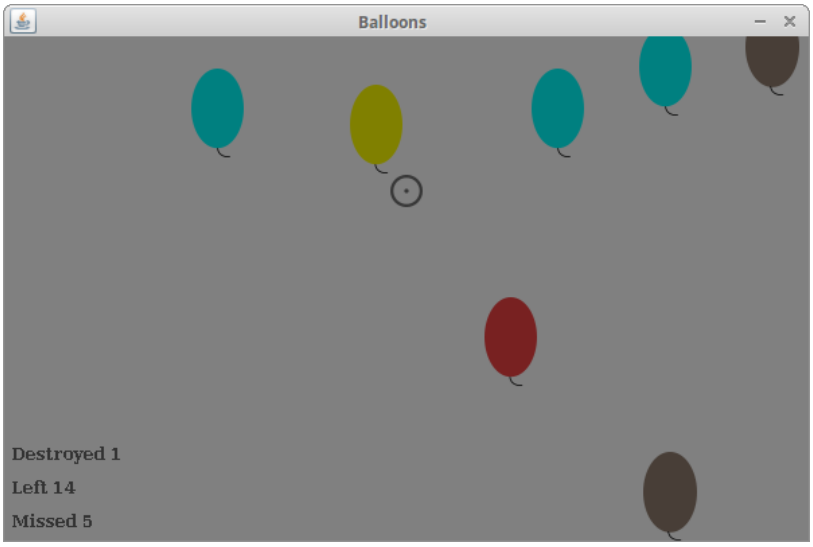

Figure 5.1: Balloons

Chapter 6

Flappy bird

Flappy Bird is a mobile game developed by a Vietnamese programmer Dong Nguyen, which became very popular in 2014. It is a simple side-scrolling game. The player controlls a bird which has to avoid pipes and the ground. The pipes move in pairs from right to left. The horizontal gaps between the pairs of pipes and their height are randomly generated. The vertical gaps between the pipes are constant.

The game is controlled with the `Space` key; it makes the bird fly. The bird also does some small rotation during its movement.

Listing 6.1: Orientation.java

```
package com.zetcode;

public enum Orientation {
    SOUTH,
    NORTH
}
```

The `Orientation` enumeration defines the orientation of the pipe.

Listing 6.2: GameConstants.java

```
package com.zetcode;

public class GameConstants {

    // Window dimensions
    public static final int WINDOW_WIDTH = 500;
    public static final int WINDOW_HEIGHT = 500;

    // Bird dimensions and velocity
    public static final int BIRD_WIDTH = 45;
    public static final int BIRD_HEIGHT = 32;
    public static final int BIRD_Y_VELOCITY = -10;

    // Pipe dimensions and gap
    public static final int PIPE_WIDTH = 66;
    public static final int PIPE_HEIGHT = 400;
    public static final int VGAP_BETWEEN_PIPES = 175;
}
```

In `GameConstants` we define the game constants. For instance, the `PIPE_WIDTH` sets the width of a pipe.

Listing 6.3: Sprite.java

```
package com.zetcode;

import java.awt.Image;

public class Sprite {

    protected int x = 0;
    protected int y = 0;
    protected Image image;

    public int getX() {

        return x;
    }

    public void setX(int x) {

        this.x = x;
    }

    public int getY() {

        return y;
    }

    public void setY(int y) {

        this.y = y;
    }

    public Image getImage() {

        return image;
    }

    public void setImage(Image image) {

        this.image = image;
    }
}
```

The `Sprite` contains common attributes for a sprite in a game. `Pipe` and `Bird` inherit from the `Sprite`.

Listing 6.4: Pipe.java

```
package com.zetcode;

import javax.swing.ImageIcon;

public class Pipe extends Sprite {
```

```
    private int speed = 2;

    private Orientation orientation;

    public Pipe(Orientation orientation) {

        initPipe(orientation);
    }

    private void initPipe(Orientation orientation) {

        this.orientation = orientation;

        String pipeOrient;

        if (orientation == Orientation.NORTH) {

            pipeOrient = "north";
        } else {

            pipeOrient = "south";
        }

        image = new ImageIcon("src/resources/pipe-"
                + pipeOrient + ".png").getImage();
    }

    public void update() {

        x -= speed;
    }

    public boolean collides(int _x, int _y, int _width, int _height) {

        int margin = 2;

        if (_x + _width - margin > x && _x + margin < x +
                GameConstants.PIPE_WIDTH) {

            if (orientation == Orientation.SOUTH && _y < y +
                    GameConstants.PIPE_HEIGHT) {

                return true;
            } else if (orientation == Orientation.NORTH &&
                    _y + _height > y) {

                return true;
            }
        }

        return false;
    }

    public Orientation getOrientation() {

        return orientation;
    }

    public int getHeight() {

        return GameConstants.PIPE_HEIGHT;
    }
```

```
    public int getWidth() {

        return GameConstants.PIPE_WIDTH;
    }

    public int getSpeed() {

        return speed;
    }

    @Override
    public String toString() {

        return "Pipe{ " + "x=" + getX() + ", y=" + getY()
                + ", orientation=" + getOrientation() + '}';
    }
}
```

The `Pipe` represents the pipe in the game. There are two kinds of pipes: orientated to the north and to the south.

```
private void initPipe(Orientation orientation) {

    this.orientation = orientation;

    String pipeOrient;

    if (orientation == Orientation.NORTH) {

        pipeOrient = "north";
    } else {

        pipeOrient = "south";
    }

    image = new ImageIcon("src/resources/pipe-"
            + pipeOrient + ".png").getImage();
}
```

We load the pipe image based on its specified orientation.

```
public void update() {

    x -= speed;
}
```

The `update()` method updates the `x` coordinate of the pipe. The pipe moves to the left, so `x` decreases.

```
public boolean collides(int _x, int _y, int _width, int _height) {

    int margin = 2;

    if (_x + _width - margin > x && _x + margin < x +
            GameConstants.PIPE_WIDTH) {

        if (orientation == Orientation.SOUTH && _y < y +
                GameConstants.PIPE_HEIGHT) {
```

```
            return true;
        } else if (orientation == Orientation.NORTH &&
                _y + _height > y) {

            return true;
        }
    }

    return false;
}
```

The `collides()` method checks if a north or a south pipe collides with the bird.

Listing 6.5: Bird.java

```
package com.zetcode;

import java.awt.geom.AffineTransform;
import javax.swing.ImageIcon;

public class Bird extends Sprite {

    private double yvel;
    private double gravity;
    private double rotation;

    public Bird() {

        initBird();
    }

    private void initBird() {

        x = 100;
        y = 150;

        yvel = 0;
        gravity = 0.9;
        rotation = 0.0;

        image = new ImageIcon("src/resources/bird.png").getImage();
    }

    public void update() {

        yvel += gravity;
        y += (int) yvel;
    }

    public AffineTransform getTransform() {

        rotation = (90 * (yvel + 20) / 20) - 90;
        rotation = rotation * Math.PI / 180;

        if (rotation > Math.PI / 2) {

            rotation = Math.PI / 2;
        }

        var aft = new AffineTransform();
        aft.translate(x + GameConstants.BIRD_WIDTH / 2,
```

```
                y + GameConstants.BIRD_HEIGHT / 2);
        aft.rotate(rotation);
        aft.translate(-GameConstants.BIRD_WIDTH / 2,
                -GameConstants.BIRD_HEIGHT / 2);

        return aft;
    }

    public void setYVelocity(int val) {

        yvel = val;
    }

    public int getWidth() {

        return GameConstants.BIRD_WIDTH;
    }

    public int getHeight() {

        return GameConstants.BIRD_HEIGHT;
    }
}
```

`Bird` represents the bird object.

```
private double yvel;
private double gravity;
private double rotation;
```

For the bird sprite, we have three important variables to work with: y-velocity, gravity, and rotation.

```
public void update() {

    yvel += gravity;
    y += (int) yvel;
}
```

The `update()` method increases the y-velocity over time and the value increases the `y` coordinate of the bird. As a consequence, the bird is dragged to the bottom. The make the bird fly, the player repeatedly presses the `Space` key.

```
public AffineTransform getTransform() {

    rotation = (90 * (yvel + 20) / 20) - 90;
    rotation = rotation * Math.PI / 180;

    if (rotation > Math.PI / 2) {

        rotation = Math.PI / 2;
    }

    var aft = new AffineTransform();
    aft.translate(x + GameConstants.BIRD_WIDTH / 2,
            y + GameConstants.BIRD_HEIGHT / 2);
    aft.rotate(rotation);
    aft.translate(-GameConstants.BIRD_WIDTH / 2,
            -GameConstants.BIRD_HEIGHT / 2);
```

```
    return aft;
}
```

The `AffineTransform` is computed; it rotates the bird. The rotation is an angle measured in radians. The value is called *theta*. It increases with the increase of the y-velocity. The `AffineTransform` is passed as a second parameter to the `Graphics2D's drawImage()` method.

Listing 6.6: Board.java

```
package com.zetcode;

import javax.swing.ImageIcon;
import javax.swing.JPanel;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.RenderingHints;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.util.ArrayList;
import java.util.List;
import java.util.SplittableRandom;
import javax.swing.Timer;

public class Board extends JPanel {

    private Timer timer;
    private boolean paused;

    private Bird bird;
    private List<List<Pipe>> pipes;

    private int score;
    private boolean gameOver;
    private boolean started;

    private Image bckImg;

    int count;

    public Board() {

        initBoard();
    }

    private void initBoard() {

        initGame();

        setPreferredSize(new Dimension(GameConstants.WINDOW_WIDTH,
                GameConstants.WINDOW_HEIGHT));
        loadBackgroundImage();

        setFocusable(true);
```

```
        addKeyListener(new TAdapter());
    }

    private void loadBackgroundImage() {

        var path = "src/resources/background.png";
        bckImg = new ImageIcon(path).getImage();
    }

    private void initGame() {

        paused = false;
        started = false;
        gameOver = false;

        score = 0;
        count = 0;

        bird = new Bird();

        initPipes();
        startTimer();
    }

    private void initPipes() {

        pipes = new ArrayList<>();
        var pair1 = new ArrayList<Pipe>();

        var pipe1 = new Pipe(Orientation.SOUTH);
        var pipe2 = new Pipe(Orientation.NORTH);

        pair1.add(pipe1);
        pair1.add(pipe2);

        pipes.add(pair1);

        var pipe3 = new Pipe(Orientation.SOUTH);
        var pipe4 = new Pipe(Orientation.NORTH);

        var pair2 = new ArrayList<Pipe>();

        pair2.add(pipe3);
        pair2.add(pipe4);

        pipes.add(pair2);

        var pipe5 = new Pipe(Orientation.SOUTH);
        var pipe6 = new Pipe(Orientation.NORTH);

        var pair3 = new ArrayList<Pipe>();

        pair3.add(pipe5);
        pair3.add(pipe6);
        pipes.add(pair3);

        generatePipePositions(pair1);
        generatePipePositions(pair2);
        generatePipePositions(pair3);
    }

    private void startTimer() {
```

```
        int fps = 60;
        int period = 1000 / fps;

        timer = new Timer(period, new GameCycle());
        timer.start();
    }

    private void generatePipePositions(List<Pipe> pair) {

        var sr = new SplittableRandom();

        int south_pipe_y = -(sr.nextInt(120)) -
                GameConstants.PIPE_HEIGHT / 2;
        int north_pipe_y = south_pipe_y + GameConstants.PIPE_HEIGHT +
                GameConstants.VGAP_BETWEEN_PIPES;

        int random_gap = sr.nextInt(0, 50);
        int FIXED_GAP = 70;

        int pipe_x = GameConstants.WINDOW_WIDTH +
                count * (GameConstants.PIPE_WIDTH + 20) +
                count * FIXED_GAP + random_gap;

        count++;

        if (count % 3 == 0) {

            count = 0;
        }

        for (Pipe pipe : pair) {

            if (pipe.getOrientation() == Orientation.SOUTH) {

                pipe.setX(pipe_x);
                pipe.setY(south_pipe_y);
            } else {

                pipe.setX(pipe_x);
                pipe.setY(north_pipe_y);
            }
        }
    }

    private void doGameCycle() {

        update();
        repaint();
    }

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);

        var g2d = (Graphics2D) g;

        var rh = new RenderingHints(
                RenderingHints.KEY_ANTIALIASING,
                RenderingHints.VALUE_ANTIALIAS_ON);

        rh.put(RenderingHints.KEY_RENDERING,
```

```
                RenderingHints.VALUE_RENDER_QUALITY);

        g2d.setRenderingHints(rh);

        if (!isGameOver()) {

            drawBackground(g2d);
            drawPipes(g2d);
            drawBird(g2d);
        }

        g2d.setColor(Color.BLACK);

        if (!isStarted()) {

            g2d.setFont(new Font("Helvetica", Font.BOLD, 20));
            g2d.drawString("Press SPACE to start", 150, 240);
        } else {

            g2d.setFont(new Font("Helvetica", Font.BOLD, 24));
            g2d.drawString(Integer.toString(getScore()), 10, 465);
        }

        if (isGameOver()) {

            g2d.setColor(Color.gray);
            g2d.fillRect(0, 0, 500, 500);

            g2d.setColor(Color.black);
            g2d.setFont(new Font("Helvetica", Font.BOLD, 20));

            var score_msg = String.format("Score: %d", score);
            g2d.drawString(score_msg, 150, 200);
            g2d.drawString("Press R to restart", 150, 240);
        }

        Toolkit.getDefaultToolkit().sync();
    }

    private void update() {

        if (gameOver || !started || paused) {

            return;
        }

        bird.update();

        movePipes();
        checkCollisions();
    }

    private void drawBackground(Graphics2D g2d) {

        g2d.drawImage(bckImg, 0, 0, null);
    }

    private void drawPipes(Graphics2D g2d) {

        for (List<Pipe> pair : pipes) {

            for (Pipe pipe : pair) {
```

```
                g2d.drawImage(pipe.getImage(), pipe.getX(),
                        pipe.getY(), null);
            }
        }
    }

    private void drawBird(Graphics2D g2d) {

        g2d.drawImage(bird.getImage(), bird.getTransform(), null);
    }

    private void movePipes() {

        for (List<Pipe> pair : pipes) {

            for (Pipe pipe : pair) {

                if (pipe.getOrientation() == Orientation.NORTH
                        && pipe.getX() + 3 * pipe.getWidth() < 0) {

                    generatePipePositions(pair);
                } else {

                    pipe.update();
                }
            }
        }
    }

    private void checkCollisions() {

        // pipe collision
        for (List<Pipe> pair : pipes) {

            for (Pipe pipe : pair) {

                int dist = pipe.getX() + pipe.getWidth() - bird.getX();

                if (pipe.collides(bird.getX(), bird.getY(),
                        bird.getWidth(), bird.getHeight())) {

                    gameOver = true;
                    timer.stop();
                } else if (dist > -pipe.getSpeed() && dist <= 0
                      && pipe.getOrientation() == Orientation.SOUTH) {

                    score++;
                }
            }
        }

        // ground collision
        if (bird.getY() + bird.getHeight()
                > GameConstants.WINDOW_HEIGHT) {

            gameOver = true;
            timer.stop();
        }
    }

    private boolean isGameOver() {
```

```
        return gameOver;
    }

    private int getScore() {

        return score;
    }

    private boolean isPaused() {

        return paused;
    }

    private void setPaused(boolean paused) {

        this.paused = paused;
    }

    private boolean isStarted() {

        return started;
    }

    private void setStarted(boolean started) {

        this.started = started;
    }

    public class TAdapter extends KeyAdapter {

        @Override
        public void keyPressed(KeyEvent e) {

            int key = e.getKeyCode();

            if (!isStarted() && key == KeyEvent.VK_SPACE) {

                setStarted(true);
            }

            if (key == KeyEvent.VK_SPACE) {

                bird.setYVelocity(GameConstants.BIRD_Y_VELOCITY);
            }

            if (key == KeyEvent.VK_P) {

                setPaused(!isPaused());
            }

            if (key == KeyEvent.VK_R) {

                initGame();
            }
        }
    }

    private class GameCycle implements ActionListener {

        @Override
        public void actionPerformed(ActionEvent e) {
```

```
            doGameCycle();
        }
    }
}
```

In `Board` we have the logic of the game.

```
private List<List<Pipe>> pipes;
```

The pipes are placed in a list of lists. The pipes are stored in pairs.

```
int count;
```

The `count` variable is used to help calculate the gaps between the pairs of pipes.

```
private void initPipes() {

    pipes = new ArrayList<>();
    var pair1 = new ArrayList<Pipe>();
...
```

In the `initPipes()`, we initiate the list of pairs of pipes.

```
var pipe1 = new Pipe(Orientation.SOUTH);
var pipe2 = new Pipe(Orientation.NORTH);

pair1.add(pipe1);
pair1.add(pipe2);

pipes.add(pair1);
```

A pair of pipes is created and added to the list. There are three pairs of pipes in the game. They move from right to the left. When a pair of pipes moves past the board, their coordinates change so that they reappear on the right again.

```
generatePipePositions(pair1);
generatePipePositions(pair2);
generatePipePositions(pair3);
```

The `generatePipePositions()` method generates the coordinates for the pipes.

```
private void generatePipePositions(List<Pipe> pair) {

    var sr = new SplittableRandom();

    int south_pipe_y = -(sr.nextInt(120)) -
            GameConstants.PIPE_HEIGHT / 2;
    int north_pipe_y = south_pipe_y + GameConstants.PIPE_HEIGHT +
            GameConstants.VGAP_BETWEEN_PIPES;
...
```

Using the `SplittableRandom` class, we create variable heights of pipes. The `GameConstants.VGAP_BETWEEN_PIPES` constant is a fixed vertical space between the north and south pipe.

```
int random_gap = sr.nextInt(0, 50);
int FIXED_GAP = 70;
```

```
int pipe_x = GameConstants.WINDOW_WIDTH +
        count*(GameConstants.PIPE_WIDTH+20) +
        count*FIXED_GAP + random_gap;

count++;

if (count % 3 == 0) {

    count = 0;
}
```

In these lines, we calculate variable horizontal distances between the pipes. There is some fixed gap and some random gap between them.

```
for (Pipe pipe : pair) {

    if (pipe.getOrientation() == Orientation.SOUTH) {

        pipe.setX(pipe_x);
        pipe.setY(south_pipe_y);
    } else {

        pipe.setX(pipe_x);
        pipe.setY(north_pipe_y);
    }
}
```

We set the generated `x` and `y` values to the pipes.

```
private void doGameCycle() {

    update();
    repaint();
}
```

The game cycle consists of two steps: the game is updated and the board is repainted.

```
if (!isGameOver()) {

    drawBackground(g2d);
    drawPipes(g2d);
    drawBird(g2d);
}
```

The drawing is divided into three steps: we draw the background, the pipes, and the bird.

```
if (!isStarted()) {

    g2d.setFont(new Font("Helvetica", Font.BOLD, 20));
    g2d.drawString("Press SPACE to start", 150, 240);
} else {

    g2d.setFont(new Font("Helvetica", Font.BOLD, 24));
    g2d.drawString(Integer.toString(getScore()), 10, 465);
}

if (isGameOver()) {
```

```
    g2d.setColor(Color.gray);
    g2d.fillRect(0, 0, 500, 500);

    g2d.setColor(Color.black);
    g2d.setFont(new Font("Helvetica", Font.BOLD, 20));

    var score_msg = String.format("Score: %d", score);
    g2d.drawString(score_msg, 150, 200);
    g2d.drawString("Press R to restart", 150, 240);
}
```

Depending on the state of the game, we draw the messages and the score. There is an initial screen and a game over screen.

```
private void update() {

    if (gameOver || !started || paused) {

        return;
    }

    bird.update();

    movePipes();
    checkCollisions();
}
```

In the `update()` method, we update the velocity and the `y` coordinate of the bird, move the pipes, and check for collisions.

```
private void movePipes() {

    for (List<Pipe> pair : pipes) {

        for (Pipe pipe : pair) {

            if (pipe.getOrientation() == Orientation.NORTH
                    && pipe.getX() + 3 * pipe.getWidth() < 0) {

                generatePipePositions(pair);
            } else {

                pipe.update();
            }
        }
    }
}
```

In the `movePipes()` method we update the pipes by decreasing their x coordinates. If the pipes pass the left border of the board, a new pair is automatically generated.

```
private void checkCollisions() {
```

The `checkCollisions()` checks for collisions in the game.

```
// pipe collision
for (List<Pipe> pair : pipes) {
```

```
    for (Pipe pipe : pair) {

        int dist = pipe.getX() + pipe.getWidth() - bird.getX();

        if (pipe.collides(bird.getX(), bird.getY(),
                bird.getWidth(), bird.getHeight())) {

            gameOver = true;
            timer.stop();
        } else if (dist > -pipe.getSpeed() && dist <= 0
                && pipe.getOrientation() == Orientation.SOUTH) {

            score++;
        }
    }
}
```

Inside a for loop, we check if any of the pipes collides with the bird. If there is a collision, we set the `gameOver` variable to `true` and kill the timer. If the bird passes a pair of pipes, the score is incremented.

```
// ground collision
if (bird.getY() + bird.getHeight()
        > GameConstants.WINDOW_HEIGHT) {

    gameOver = true;
    timer.stop();
}
```

We also check for ground collision.

```
public class TAdapter extends KeyAdapter {

    @Override
    public void keyPressed(KeyEvent e) {

        int key = e.getKeyCode();

        if (!isStarted() && key == KeyEvent.VK_SPACE) {

            setStarted(true);
        }

        if (key == KeyEvent.VK_SPACE) {

            bird.setYVelocity(GameConstants.BIRD_Y_VELOCITY);
        }

        if (key == KeyEvent.VK_P) {

            setPaused(!isPaused());
        }

        if (key == KeyEvent.VK_R) {

            initGame();
        }
    }
}
```

The game is controlled with keys. The `Space` starts the game and increases the

velocity of the bird. The game is paused with the `P` key and restarted with the `R` key.

Listing 6.7: FlappyBird.java

```
package com.zetcode;

import java.awt.EventQueue;
import javax.swing.JFrame;

/**
 * Java Flappy bird game
 *
 * Author: Jan Bodnar
 * Website: http://zetcode.com
 */

public class FlappyBird extends JFrame {


    public FlappyBird() {

        initUI();
    }

    private void initUI() {

        var panel = new Board();
        add(panel);

        setResizable(false);
        pack();

        setTitle("Flappy Bird");
        setLocationRelativeTo(null);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public static void main(String[] args) {

        EventQueue.invokeLater(() -> {

            var game = new FlappyBird();
            game.setVisible(true);
        });
    }
}
```

`FlappyBird` initiates the Flappy bird game.

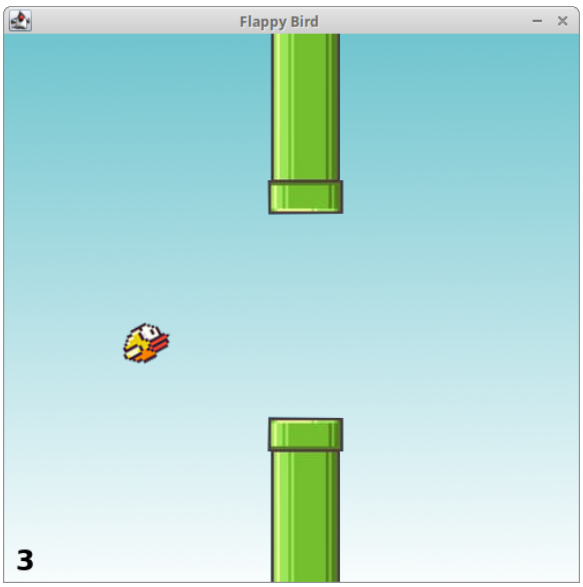

Figure 6.1: Flappy bird

Chapter 7

Cannon

Cannon is a simple 2D game in which the player shoots at planes with a cannon. He is controlling the cannon position with cursor keys. With the `Space` key, he controlls the power of the shot. The current power is shown in the indicator, which is displayed in the top-left corner of the window. In order to hit the plane, he must choose the correct angle and power. The game finishes if three planes pass the left border of the window.

In the Cannon game, we have three sprites: the cannon, the plane, and the projectile.

Listing 7.1: Sprite.java

```
package com.zetcode;

import java.awt.Image;

public class Sprite {

    protected int imageWidth = 0;
    protected int imageHeight = 0;

    protected double x = 0;
    protected double y = 0;

    protected Image image;


    public int getImageWidth() {

        return imageWidth;
    }

    public int getImageHeight() {

        return imageHeight;
    }

    public double getX() {

        return x;
    }
```

```
    public double getY() {

        return y;
    }

    public Image getImage() {

        return image;
    }
}
```

The `Sprite` represents a sprite in the game. It contains common attributes including the `x` and `y` coordinates, `imageWidth` and `imageHeight`, and `image`.

Listing 7.2: GameCons.java

```
package com.zetcode;

public class GameCons {

    public static final int WINDOW_WIDTH = 1000;
    public static final int WINDOW_HEIGHT = 500;
    public static final int GROUND_THRESHOLD = 455;

    public static final int CANNON_INIT_X = 60;
    public static final int CANNON_INIT_Y = 450;

    public static final int BALL_INIT_X = 65;
    public static final int BALL_INIT_Y = 450;

    public static final int INDICATOR_INIT_X = 120;
    public static final int INDICATOR_INIT_Y = 10;

    public static final int MAX_CANNON_ANGLE = 90;
    public static final int MIN_CANNON_ANGLE = 0;

    public static final int MAX_TIME_DELTA = 2000;
    public static final int MIN_TIME_DELTA = 100;
    public static final int TIME_POWER_RATIO = 50;

    public static final float INDICATOR_RATIO = MAX_TIME_DELTA /
            TIME_POWER_RATIO;
}
```

We have the game constants.

```
public static final int GROUND_THRESHOLD = 455;
```

The `GROUND_THRESHOLD` constant is used for measuring when the projectile hits the ground.

```
public static final int CANNON_INIT_X = 60;
public static final int CANNON_INIT_Y = 450;
```

These are initial coordinates of the cannon.

```
public static final int MAX_TIME_DELTA = 2000;
public static final int MIN_TIME_DELTA = 100;
```

```
public static final int TIME_POWER_RATIO = 50;
```

These values are used to calculate the power of the shot.

```
public static final float INDICATOR_RATIO = MAX_TIME_DELTA /
        TIME_POWER_RATIO;
```

This value is used to calculate the filling of the power indicator.

Listing 7.3: GameMode.java

```
package com.zetcode;

public enum GameMode {

    GAME_ON,
    GAME_PAUSED,
    GAME_OVER
}
```

`GameMode` is a simple enumeration which holds three states of the game: on, paused, and over.

Listing 7.4: Plane.java

```
package com.zetcode;

import java.awt.Rectangle;
import javax.swing.ImageIcon;

public class Plane extends Sprite {

    public Plane(int yPos) {

        initPlane(yPos);
    }

    private void initPlane(int yPos) {

        loadImage();

        y = yPos;
        x = 1000;
    }

    private void loadImage() {

        var filePath = "src/resources/plane.png";

        image = new ImageIcon(filePath).getImage();
        imageWidth = image.getWidth(null);
        imageHeight = image.getHeight(null);
    }

    public Rectangle getBounds() {

        return new Rectangle((int) x - imageWidth / 2,
                (int) y - imageHeight / 4, imageWidth,
                imageHeight / 2);
```

```
    }

    public void update() {

        x -= 1;
    }
}
```

`Plane` is the enemy plane that we shoot at.

```
private void initPlane(int yPos) {

    loadImage();

    y = yPos;
    x = 1000;
}
```

The initial `x` coordinate is fixed for all planes; it is located beyond the border of the window. The `y` coordinate is randomly picked at the creation of the plane.

```
public Rectangle getBounds() {

    return new Rectangle((int) x - imageWidth / 2,
            (int) y - imageHeight / 4, imageWidth,
            imageHeight / 2);
}
```

We get the bounds of the plane. This is used for collision detection.

```
public void update() {

    x -= 1;
}
```

The plane is moving by decreasing the `x` coordinate value.

Listing 7.5: Projectile.java

```
package com.zetcode;

import javax.swing.ImageIcon;
import java.awt.Rectangle;

import static java.lang.Math.cos;
import static java.lang.Math.sin;
import static java.lang.Math.sqrt;
import static java.lang.Math.toRadians;

public class Projectile extends Sprite {

    private double v0 = 0;
    private double angle = 0;
    private double acceleration = 0;

    public Projectile(double angle, double energy,
                      double x_, double y_) {

        loadImages();
```

```
        initProjectile(angle, energy, x_, y_);
    }

    private void loadImages() {

        var imgPath = "src/resources/ball.png";

        image = new ImageIcon(imgPath).getImage();

        imageWidth = image.getWidth(null);
        imageHeight = image.getHeight(null);
    }

    private void initProjectile(double cannon_angle,
            double cannon_power, double x, double y) {

        this.x = x;
        this.y = y;

        angle = toRadians(cannon_angle);
        double energy = cannon_power * 10;
        double mass = 4;

        v0 = sqrt((energy * 2) / mass);
    }

    public Rectangle getBounds() {

        return new Rectangle((int) x - imageWidth / 2,
                (int) y - imageHeight / 2,
                imageWidth, imageHeight);
    }

    public void update() {

        acceleration = acceleration + 0.09;

        double vx = v0 * cos(angle);
        double vy = v0 * sin(angle) - acceleration;

        x = x + vx;
        y = y - vy;
    }
}
```

`Projectile` is the ball that we shoot at the moving planes.

```
private double v0 = 0;
private double angle = 0;
private double acceleration = 0;
```

The `v0` is the velocity of the projectile. The `angle` is the angle at which the projectile was shot from the cannon. The `acceleration` is the rate of change of the velocity in the y-direction over time.

```
private void initProjectile(double cannon_angle,
        double cannon_power, double x, double y) {

    this.x = x;
    this.y = y;
```

```
    angle = toRadians(cannon_angle);
    double energy = cannon_power * 10;
    double mass = 4;

    v0 = sqrt((energy * 2) / mass);
}
```

When we initiate the projectile, we store its `x` and `y` coordinates and transform the angle value to radians. The `energy` is the power of the shot; the longer we hold the fire key, the greater the energy of the projectile becomes. The velocity of the projectile is computed from the following equation:

$$e = \frac{1}{2}mv^2 \tag{7.1}$$

The kinetic energy is directly proportional to the mass of the object and to the square of its velocity.

```
public void update() {

    acceleration = acceleration + 0.09;

    double vx = v0 * cos(angle);
    double vy = v0 * sin(angle) - acceleration;

    x = x + vx;
    y = y - vy;
}
```

In the `update()` method, we calculate the horizontal and vertical displacement of the projectile. The acceleration represents the gravity pulling the projectile to the ground.

Listing 7.6: Board.java

```
package com.zetcode;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.Rectangle;
import java.awt.RenderingHints;
import java.awt.Toolkit;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.util.List;
import javax.swing.ImageIcon;
import javax.swing.JPanel;
import javax.swing.Timer;


public class Board extends JPanel {
```

```
    private int score = 0;
    private int missedPlanes = 0;
    private int destroyedPlanes = 0;

    private Image background_image;
    private Image cannon1_image;
    private Image cannon2_image;

    private GameMode gameMode = GameMode.GAME_ON;
    private List<Projectile> projectiles = new ArrayList<>();
    private List<Plane> planes = new ArrayList<>();

    private int cannonAngleValue = 60;
    private int cannonPowerValue = 0;

    private int delta = 0;
    private long startTime;

    private boolean startedIndicator = false;
    private boolean paused = false;

    private TAdapter adapter;
    private Timer timer;

    public Board() {

        initBoard();
        loadImages();
    }

    private void initBoard() {

        setFocusable(true);

        setPreferredSize(new Dimension(GameCons.WINDOW_WIDTH,
                GameCons.WINDOW_HEIGHT));


        adapter = new TAdapter();
        addKeyListener(adapter);

        int fps = 60;

        timer = new Timer(1000 / fps, new GameCycle());
        timer.start();
    }

    private void loadImages() {

        var fileBack = "src/resources/background.png";
        var fileCan1 = "src/resources/cannon1.png";
        var fileCan2 = "src/resources/cannon2.png";

        background_image = new ImageIcon(fileBack).getImage();
        cannon1_image = new ImageIcon(fileCan1).getImage();
        cannon2_image = new ImageIcon(fileCan2).getImage();
    }

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

```
        var g2d = (Graphics2D) g;

        var rh = new RenderingHints(
                RenderingHints.KEY_ANTIALIASING,
                RenderingHints.VALUE_ANTIALIAS_ON);

        rh.put(RenderingHints.KEY_RENDERING,
                RenderingHints.VALUE_RENDER_QUALITY);

        g2d.setRenderingHints(rh);
        g2d.drawImage(background_image, 0, 0, this);

        if (gameMode == GameMode.GAME_ON) {

            drawStats(g2d);
            drawProjectiles(g2d);
            drawPlanes(g2d);
            drawCannon(g2d);
            drawCannonPowerIndicator(g2d);
        }

        if (gameMode == GameMode.GAME_OVER) {

            drawGameOver(g2d);
        }

        if (gameMode == GameMode.GAME_PAUSED) {

            drawGamePaused(g2d);
        }

        Toolkit.getDefaultToolkit().sync();
    }

    private void drawPlanes(Graphics2D g2d) {

        for (int i = 0; i < planes.size(); i++) {

            Plane plane = planes.get(i);

            g2d.drawImage(plane.getImage(),
                    (int) (plane.getX() - plane.getImageWidth() / 2),
                    (int) (plane.getY() - plane.getImageHeight() / 2),
                    this);
        }
    }

    private void drawStats(Graphics2D g2d) {

        g2d.setFont(new Font("Serif", Font.BOLD, 12));
        g2d.drawString("Destroyed: " + destroyedPlanes, 5, 20);
        g2d.drawString("Escaped: " + missedPlanes, 5, 40);
        g2d.drawString("Score: " + score, 5, 60);
    }

    private void drawProjectiles(Graphics2D g2d) {

        for (int i = 0; i < projectiles.size(); i++) {

            Projectile proj = projectiles.get(i);

            g2d.drawImage(proj.getImage(),
```

```
                (int) (proj.getX() - proj.getImageWidth() / 2),
                (int) (proj.getY() - proj.getImageHeight() / 2),
                this);
        }
    }

    private void drawGameOver(Graphics2D g2d) {

        g2d.setFont(new Font("Serif", Font.BOLD, 70));
        g2d.drawString("Game Over", 320, 90);

        g2d.setFont(new Font("Arial", Font.BOLD, 40));
        g2d.drawString("Score: " + score, 410, 150);
    }

    private void drawCannonPowerIndicator(Graphics2D g2d) {

        g2d.setColor(new Color(80, 80, 80));

        // resetting the previous translation
        g2d.translate(-GameCons.CANNON_INIT_X,
                -GameCons.CANNON_INIT_Y);

        g2d.translate(GameCons.INDICATOR_INIT_X,
                GameCons.INDICATOR_INIT_Y);
        g2d.drawRect(0, 0, 100, 10);

        int w = (int) ((getCanonPowerValue() /
                GameCons.INDICATOR_RATIO) * 100);
        g2d.fillRect(0, 0, w, 10);
    }

    private void drawCannon(Graphics2D g2d) {

        g2d.translate(GameCons.CANNON_INIT_X,
                GameCons.CANNON_INIT_Y);
        g2d.rotate(Math.toRadians(-getCannonAngleValue()));

        g2d.drawImage(cannon2_image, -13, -13, this);
        g2d.rotate(Math.toRadians(getCannonAngleValue()));
        g2d.drawImage(cannon1_image, -36, -3, this);
    }

    private void drawGamePaused(Graphics2D g2d) {

        g2d.setFont(new Font("Serif", Font.BOLD, 60));
        g2d.drawString("Game Paused", 310, 150);
    }


    private void doGameCycle() {


        if (isPaused()) {

            gameMode = GameMode.GAME_PAUSED;
        } else {

            gameMode = GameMode.GAME_ON;
        }

        if (gameMode == GameMode.GAME_ON) {
```

```
            if (Math.random() < (0.004 + (destroyedPlanes * 0.0003))) {

                int y = 20 + (int) (Math.random() * 350);
                planes.add(new Plane(y));
            }

            for (int i = 0; i < projectiles.size(); i++) {

                Projectile projectile = projectiles.get(i);
                projectile.update();
            }

            for (int i = 0; i < planes.size(); i++) {

                Plane plane = planes.get(i);
                plane.update();
            }

            checkPlanesHit();
            checkEscapedPlanes();
            checkProjectileHitGround();

            int maxMissedPlanes = 3;

            if (missedPlanes >= maxMissedPlanes) {

                gameMode = GameMode.GAME_OVER;
                timer.stop();
                removeKeyListener(adapter);
            }

            cannonAngleValue += delta;

            if (cannonAngleValue > GameCons.MAX_CANNON_ANGLE) {

                cannonAngleValue = GameCons.MAX_CANNON_ANGLE;
            }

            if (cannonAngleValue < GameCons.MIN_CANNON_ANGLE) {

                cannonAngleValue = GameCons.MIN_CANNON_ANGLE;
            }
        }

        repaint();
    }

    private void checkPlanesHit() {

        for (int i = 0; i < projectiles.size(); i++) {

            Projectile projectile = projectiles.get(i);

            Rectangle rBall = projectile.getBounds();

            for (int u = 0; u < planes.size(); u++) {

                Plane plane = planes.get(u);
                Rectangle rPlane = plane.getBounds();

                if (rPlane.intersects(rBall)) {
```

```
                planes.remove(plane);
                projectiles.remove(projectile);
                destroyedPlanes++;

                score += 20;
            }
        }
    }
}

private void checkProjectileHitGround() {

    for (int i = 0; i < projectiles.size(); i++) {

        Projectile projectile = projectiles.get(i);

        if (projectile.getY() > (GameCons.GROUND_THRESHOLD
                + Math.random() * 10) ||
                projectile.getX() > GameCons.WINDOW_WIDTH) {

            projectiles.remove(projectile);
        }
    }
}

private void checkEscapedPlanes() {

    for (int i = 0; i < planes.size(); i++) {

        Plane plane = planes.get(i);

        if (plane.getX() < -plane.getImageWidth()) {

            planes.remove(plane);

            missedPlanes++;
        }
    }
}

private int getCannonAngleValue() {

    return cannonAngleValue;
}

private int getCanonPowerValue() {

    if (startedIndicator) {

        long now = System.currentTimeMillis();
        long timeDelta = now - startTime;

        if (timeDelta > GameCons.MAX_TIME_DELTA) {

            timeDelta = GameCons.MAX_TIME_DELTA;
        }

        if (timeDelta < GameCons.MIN_TIME_DELTA) {

            timeDelta = GameCons.MIN_TIME_DELTA;
        }
```

```
            cannonPowerValue = (int) timeDelta /
                    GameCons.TIME_POWER_RATIO;
        }

        return cannonPowerValue;
    }

    private boolean isPaused() {

        return paused;
    }

    private void fire() {

        projectiles.add(new Projectile(getCannonAngleValue(),
                getCanonPowerValue(), GameCons.BALL_INIT_X,
                GameCons.BALL_INIT_Y));
    }

    private class TAdapter extends KeyAdapter {

        @Override
        public void keyPressed(KeyEvent e) {

            int key = e.getKeyCode();

            if ((key == KeyEvent.VK_SPACE)) {

                if (!startedIndicator) {

                    startTime = System.currentTimeMillis();
                }

                startedIndicator = true;
            }

            if ((key == KeyEvent.VK_P)) {

                paused = true;
            }

            if ((key == KeyEvent.VK_R)) {

                paused = false;
            }

            if ((key == KeyEvent.VK_UP)) {

                delta = 1;
            }

            if ((key == KeyEvent.VK_DOWN)) {

                delta = -1;
            }
        }

        @Override
        public void keyReleased(KeyEvent e) {

            int key = e.getKeyCode();
```

```
            if (key == KeyEvent.VK_UP ||
                    key == KeyEvent.VK_DOWN) {

                delta = 0;
            }

            if (key == KeyEvent.VK_SPACE) {

                startedIndicator = false;

                fire();
                cannonPowerValue = 0;
            }
        }
    }

    private class GameCycle implements ActionListener {

        @Override
        public void actionPerformed(ActionEvent e) {

            doGameCycle();
        }
    }
}
```

This is the board of the game.

```
private int score = 0;
private int missedPlanes = 0;
private int destroyedPlanes = 0;
```

The `score` variable holds the score of the game; the player receives twenty points for each destroyed plane. All planes that go beyond the left border of the window are stored in the `missedPlanes` variable. All destroyed planes are stored in the `destroyedPlanes` variable.

```
private Image background_image;
private Image canon1_image;
private Image canon2_image;
```

These three variables are used for a background and for the cannon; the cannon consists of two images: the barrel and the base structure. We need a separate barrel because it is moving.

```
private GameMode gameMode = GameMode.GAME_ON;
private List<Projectile> projectiles = new ArrayList<>();
private List<Plane> planes = new ArrayList<>();
```

At the beginning, the game mode is on. The `projectiles` and the `planes` lists hold all the projectiles and planes in the game.

```
private int cannonAngleValue = 60;
private int cannonPowerValue = 0;
```

The `cannonAngleValue` determines the angle at which the projectile is shot. The projectile then moves along a parabola. The `cannonPowerValue` determines the

energy of the shot. The power is set with the fire key. The power value is displayed in the indicator in the upper-left corner of the board.

```
private int delta = 0;
private long startTime;
```

The `delta` variable is used to calculate the angle of the cannon. The `startTime` variable marks the start time of the cannon shot. The longer we hold the key, the more energy the projectile has.

```
private boolean startedIndicator = false;
private boolean paused = false;
```

The `startedIndicator` variable determines whether we have started firing a projectile. The `paused` variable determines whether the game is paused.

```
private void initBoard() {

    setFocusable(true);

    setPreferredSize(new Dimension(GameCons.WINDOW_WIDTH,
            GameCons.WINDOW_HEIGHT));

    adapter = new TAdapter();
    addKeyListener(adapter);

    int fps = 60;

    timer = new Timer(1000 / fps, new GameCycle());
    timer.start();
}
```

In the `initBoard()` method, we set the focus to the board with the `setFocusable()` method. We set the preferred size for the board using the `setPreferredSize()` method. The key listener is added with the `addKeyListener()` method. Then we create and start a timer.

```
private void loadImages() {

    var fileBack = "src/resources/background.png";
    var fileCan1 = "src/resources/cannon1.png";
    var fileCan2 = "src/resources/cannon2.png";

    background_image = new ImageIcon(fileBack).getImage();
    cannon1_image = new ImageIcon(fileCan1).getImage();
    cannon2_image = new ImageIcon(fileCan2).getImage();
}
```

In the `loadImages()` method, we load three images. One is one background image and two cannon images.

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

    var g2d = (Graphics2D) g;

    var rh = new RenderingHints(
            RenderingHints.KEY_ANTIALIASING,
```

```
            RenderingHints.VALUE_ANTIALIAS_ON);

    rh.put(RenderingHints.KEY_RENDERING,
            RenderingHints.VALUE_RENDER_QUALITY);

    g2d.setRenderingHints(rh);
    g2d.drawImage(background_image, 0, 0, this);
...
```

In the `paintComponent()` method, we set the rendering hints to get a higher quality rendering. We draw the background image on the whole window with the `drawImage()` method.

```
...
if (gameMode == GameMode.GAME_ON) {

    drawStats(g2d);
    drawProjectiles(g2d);
    drawPlanes(g2d);
    drawCannon(g2d);
    drawCannonPowerIndicator(g2d);
}

if (gameMode == GameMode.GAME_OVER) {

    drawGameOver(g2d);
}

if (gameMode == GameMode.GAME_PAUSED) {

    drawGamePaused(g2d);
}

Toolkit.getDefaultToolkit().sync();
}
```

If the game mode is on, we use five methods to draw game statistics, flying projectiles and planes, the cannon and the cannot power indicator. If the game is over, we draw the "Game over" message. If the game is paused, we draw the "Game Paused" message.

```
private void drawPlanes(Graphics2D g2d) {

    for (int i = 0; i < planes.size(); i++) {

        Plane plane = planes.get(i);

        g2d.drawImage(plane.getImage(),
                (int) (plane.getX() - plane.getImageWidth() / 2),
                (int) (plane.getY() - plane.getImageHeight() / 2),
                this);
    }
}
```

In the `drawPlanes()` method, we go through the planes list and draw each of the planes on the board with the `drawImage()` method.

```
private void drawStats(Graphics2D g2d) {

    g2d.setFont(new Font("Serif", Font.BOLD, 12));
```

```
    g2d.drawString("Destroyed: " + destroyedPlanes, 5, 20);
    g2d.drawString("Escaped: " + missedPlanes, 5, 40);
    g2d.drawString("Score: " + score, 5, 60);
}
```

We draw the game statistics in the `drawStats()` method. We draw the number of the destroyed and the missed planes and the score with the `drawString()` method.

```
private void drawProjectiles(Graphics2D g2d) {

    for (int i = 0; i < projectiles.size(); i++) {

        Projectile proj = projectiles.get(i);

        g2d.drawImage(proj.getImage(),
                (int) (proj.getX() - proj.getImageWidth() / 2),
                (int) (proj.getY() - proj.getImageHeight() / 2),
                this);
    }
}
```

We draw the projectiles on the board with the `drawProjectiles()` method if there are any. We can shoot several projectiles quickly and there can be multiple flying projectiles at one moment on the board.

```
private void drawGameOver(Graphics2D g2d) {

    g2d.setFont(new Font("Serif", Font.BOLD, 70));
    g2d.drawString("Game over", 320, 90);

    g2d.setFont(new Font("Arial", Font.BOLD, 40));
    g2d.drawString("Score: " + score, 410, 150);
}
```

We draw the "Game over" message with the `drawGameOver()` method. Below the message we draw the final score.

```
private void drawCannonPowerIndicator(Graphics2D g2d) {

    g2d.setColor(new Color(80, 80, 80));

    // resetting the previous translation
    g2d.translate(-GameCons.CANNON_INIT_X,
            -GameCons.CANNON_INIT_Y);

    g2d.translate(GameCons.INDICATOR_INIT_X,
            GameCons.INDICATOR_INIT_Y);
    g2d.drawRect(0, 0, 100, 10);

    int w = (int) ((getCanonPowerValue() /
            GameCons.INDICATOR_RATIO) * 100);
    g2d.fillRect(0, 0, w, 10);
}
```

The power indicator is drawn using the `drawCannonPowerIndicator()` method. It is a small rectangle shown at the top-left corner of the window. The indicator shows the current power of the shot, which depends on how long we hold the fire key. First, we move the graphics context origin to the top-left corner, while

resetting the previous translation.

The drawing has two steps: the border and the filling of the indicator. The border is drawn with the `drawRect()` method and the filling with the `fillRect()` method. The width of the filling depends on the cannon power value, which is retrieved from the `getCanonPowerValue()` method.

```
private void drawCannon(Graphics2D g2d) {

    g2d.translate(GameCons.CANNON_INIT_X,
            GameCons.CANNON_INIT_Y);
    g2d.rotate(Math.toRadians(-getCannonAngleValue()));

    g2d.drawImage(cannon2_image, -13, -13, this);
    g2d.rotate(Math.toRadians(getCannonAngleValue()));
    g2d.drawImage(cannon1_image, -36, -3, this);
}
```

The `drawCannon()` method draws the cannon. It consists of two parts: the barrel and the base structure. The barrel is rotated depending on the angle value.

```
private void drawGamePaused(Graphics2D g2d) {

    g2d.setFont(new Font("Serif", Font.BOLD, 60));
    g2d.drawString("Game Paused", 310, 150);
}
```

When the game is paused, we draw "Game Paused" message on the board. We select a font with `setFont()` and draw the text with `drawString()`.

```
private void doGameCycle() {

    if (isPaused()) {

        gameMode = GameMode.GAME_PAUSED;
    } else {

        gameMode = GameMode.GAME_ON;
    }
...
```

In the `doGameCycle()` method, we first check if the `paused` variable is set and choose the game mode accordingly.

```
if (gameMode == GameMode.GAME_ON) {

    if (Math.random() < (0.004 + (destroyedPlanes * 0.0003))) {

        int y = 20 + (int) (Math.random() * 350);
        planes.add(new Plane(y));
    }
...
```

If the game mode is on, we first randomly decide whether to bring a new plane to the game. The higher the number of destroyed planes, the higher the probability for a new plane. The `x` value is the same for all planes (set in the `Plane` class); the `y` value is randomly chosen here.

```
for (int i = 0; i < projectiles.size(); i++) {
```

```
    Projectile projectile = projectiles.get(i);
    projectile.update();
}

for (int i = 0; i < planes.size(); i++) {

    Plane plane = planes.get(i);
    plane.update();
}
```

We go through all the projectiles and planes and update their positions.

```
checkPlanesHit();
checkEscapedPlanes();
checkProjectileHitGround();
```

In these three methods, we check for collisions.

```
int maxMissedPlanes = 3;

if (missedPlanes >= maxMissedPlanes) {

    gameMode = GameMode.GAME_OVER;
    timer.stop();
    removeKeyListener(adapter);
}
```

If we have missed more than three planes, the game is over. We stop the timer and remove the key listener.

```
cannonAngleValue += delta;

if (cannonAngleValue > GameCons.MAX_CANNON_ANGLE) {

    cannonAngleValue = GameCons.MAX_CANNON_ANGLE;
}

if (cannonAngleValue < GameCons.MIN_CANNON_ANGLE) {

    cannonAngleValue = GameCons.MIN_CANNON_ANGLE;
}
```

While checking for the minimum and maximum allowed positions, we update the `cannonAngleValue`. The value is used when drawing the barrel and firing a new projectile.

```
private void checkPlanesHit() {

    for (int i = 0; i < projectiles.size(); i++) {

        Projectile projectile = projectiles.get(i);

        Rectangle rBall = projectile.getBounds();

        for (int u = 0; u < planes.size(); u++) {

            Plane plane = planes.get(u);
            Rectangle rPlane = plane.getBounds();
```

```
            if (rPlane.intersects(rBall)) {

                planes.remove(plane);
                projectiles.remove(projectile);
                destroyedPlanes++;

                score += 20;
            }
        }
    }
}
```

In the `checkPlanesHit()` method, we check for collisions of projectiles and planes. We get the bounds of the two objects with the `getBounds()` method. The collision is detected with the `intersects()` method. In case of a collision, we remove the projectile and the plane and increase the number of destroyed planes and the score.

```
private void checkProjectileHitGround() {

    for (int i = 0; i < projectiles.size(); i++) {

        Projectile projectile = projectiles.get(i);

        if (projectile.getY() > (GameCons.GROUND_THRESHOLD
                + Math.random() * 10) ||
                projectile.getX() > GameCons.WINDOW_WIDTH) {

            projectiles.remove(projectile);
        }
    }
}
```

In the `checkProjectileHitGround()` method, we check if any projectile hits the ground threshold or the right window border. If the condition is true, the projectile is removed from the list.

```
private void checkEscapedPlanes() {

    for (int i = 0; i < planes.size(); i++) {

        Plane plane = planes.get(i);

        if (plane.getX() < -plane.getImageWidth()) {

            planes.remove(plane);

            missedPlanes++;
        }
    }
}
```

In the `checkEscapedPlanes()`, we check for escaped planes. The plane escapes if it goes past the left border of the window. If there is an escape, the plane is removed from the list and the number of missed planes is incremented. The game is finished if there are more than three missed planes.

```
private int getCanonPowerValue() {
```

```
    if (startedIndicator) {

        long now = System.currentTimeMillis();
        long timeDelta = now - startTime;

        if (timeDelta > GameCons.MAX_TIME_DELTA) {

            timeDelta = GameCons.MAX_TIME_DELTA;
        }

        if (timeDelta < GameCons.MIN_TIME_DELTA) {

            timeDelta = GameCons.MIN_TIME_DELTA;
        }

        cannonPowerValue = (int) timeDelta /
                GameCons.TIME_POWER_RATIO;
    }

    return cannonPowerValue;
}
```

The `getCanonPowerValue()` returns the power of the cannon shot. There are some minimum and maximum values of the power. The power of the shot depends on the length of the time we kept the fire key pressed.

```
private void fire() {

    projectiles.add(new Projectile(getCannonAngleValue(),
            getCanonPowerValue(), GameCons.BALL_INIT_X,
            GameCons.BALL_INIT_Y));
}
```

The `fire()` method creates a new projectile. The projectile takes the angle value, the power value, and the initial `x` and `y` coordinates as parameters.

```
private class TAdapter extends KeyAdapter {
```

The `KeyAdapter` class is a convenience class for receiving keyboard events. We will listen for key pressed and key released events. Sometimes it is necessary to listen to both kinds of events. When we fire a shot from the cannon, its power is determined from the time the `Space` key was pressed. The start time is determined in the key pressed event and the end time in the key released event.

```
@Override
public void keyPressed(KeyEvent e) {

    int key = e.getKeyCode();
...
```

In the `keyPressed()` event, we check for keys being pressed during the game.

```
if ((key == KeyEvent.VK_SPACE)) {

    if (!startedIndicator) {

        startTime = System.currentTimeMillis();
    }
```

```
    startedIndicator = true;
}
```

Here we get the start time of the `Space` key being pressed. We also set the `startedIndicator` to `true`. This is used to draw the cannon power indicator.

```
if ((key == KeyEvent.VK_P)) {

    paused = true;
}

if ((key == KeyEvent.VK_R)) {

    paused = false;
}
```

The `P` and `R` keys are used to pause and resume the game.

```
if ((key == KeyEvent.VK_UP)) {

    delta = 1;
}

if ((key == KeyEvent.VK_DOWN)) {

    delta = -1;
}
```

The cursor up and cursor down keys set the `delta` variable, which is used to move the cannon barrel.

```
@Override
public void keyReleased(KeyEvent e) {

    int key = e.getKeyCode();

    if (key == KeyEvent.VK_UP ||
            key == KeyEvent.VK_DOWN) {

        delta = 0;
    }

    if (key == KeyEvent.VK_SPACE) {

        startedIndicator = false;

        fire();
        cannonPowerValue = 0;
    }
}
```

When the cursor up or down keys are released, the `delta` value is set to `0`. The cannon barrel stops moving. Releasing the `Space` key, the `startedIndicator` is set to `false`. The power indicator will not be updated anymore. The projectile is fired with the `fire()` method. Finally, the `cannonPowerValue()` is set to 0.

```
private class GameCycle implements ActionListener {

    @Override
    public void actionPerformed(ActionEvent e) {
```

```
        doGameCycle();
    }
}
```

The `ActionListener's actionPerformed()` is periodically called by the timer. Inside the method, we call the `doGameCycle()` method, creating the game cycle.

Listing 7.7: Cannon.java

```
package com.zetcode;

import java.awt.EventQueue;
import javax.swing.JFrame;

/**
 * Java Cannon game
 *
 * Author: Jan Bodnar
 * Website: http://zetcode.com
 */

public class Cannon extends JFrame {

    public Cannon() {

        initUI();
    }

    private void initUI() {

        add(new Board());

        setResizable(false);
        pack();

        setTitle("Cannon");
        setLocationRelativeTo(null);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public static void main(String[] args) {

        EventQueue.invokeLater(() -> {

            var cannon = new Cannon();
            cannon.setVisible(true);
        });
    }
}
```

In the `Cannon` class, we set up the game.

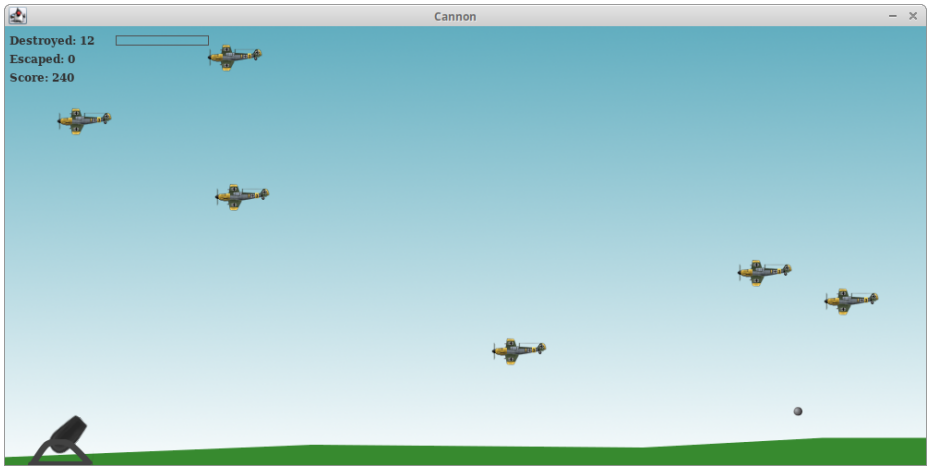

Figure 7.1: Cannon

Bibliography

1. `JDK 11 Documentation`, Oracle, 2019
2. `Filthy Rich Clients`, Chet Haase, Romain Guy 2007

Index