**Basics of Game Programming in java**

[**http://zetcode.com/tutorials/javagamestutorial/basics/**](http://zetcode.com/tutorials/javagamestutorial/basics/)

In this part of the Java 2D games tutorial, we will write about some basics needed to create games. We create a basic application, paint a donut, and display a picture.

**About**

This is Java 2D games tutorial. It is aimed at beginners. This tutorial will teach you the basics of programming 2D games in Java programming language and Swing GUI toolkit. Images used in this tutorial can be downloaded [here](http://zetcode.com/img/gfx/javagames/images.zip).

**Application**

We will show the skeleton of each of the games in this tutorial.

Board.java

package com.zetcode;

import javax.swing.JPanel;

public class Board extends JPanel {

public Board() {}

}

The Board is a panel where the game takes place.

Application.java

package com.zetcode;

import java.awt.EventQueue;

import javax.swing.JFrame;

public class Application extends JFrame {

public Application() {

initUI();

}

private void initUI() {

add(new Board());

setSize(250, 200);

setTitle("Application");

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setLocationRelativeTo(null);

}

public static void main(String[] args) {

EventQueue.invokeLater(new Runnable() {

@Override

public void run() {

Application ex = new Application();

ex.setVisible(true);

}

});

}

}

This is the entry point of the game. Here we have the main method.

add(new Board());

Here we put the Board to the center of the JFrame container.

setSize(250, 200);

This line sets the size of the window.

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

This will close the application when we click on the close button. It is not the default behaviour.

setLocationRelativeTo(null);

Passing null to the setLocationRelativeTo() method centers the window on the screen.

public static void main(String[] args) {

EventQueue.invokeLater(new Runnable() {

@Override

public void run() {

Application ex = new Application();

ex.setVisible(true);

}

});

}

We create an instance of our code example and make it visible on the screen.

**Donut**

The objects on the board are either images or are drawn with the painting tools provided by the Java 2D API. In the next example, we draw a donut shape.

Board.java

package com.zetcode;

import java.awt.BasicStroke;

import java.awt.Color;

import java.awt.Dimension;

import java.awt.Graphics;

import java.awt.Graphics2D;

import java.awt.RenderingHints;

import java.awt.geom.AffineTransform;

import java.awt.geom.Ellipse2D;

import javax.swing.JPanel;

public class Board extends JPanel {

@Override

public void paintComponent(Graphics g) {

super.paintComponent(g);

drawDonut(g);

}

private void drawDonut(Graphics g) {

Graphics2D g2d = (Graphics2D) g;

RenderingHints rh

= new RenderingHints(RenderingHints.KEY\_ANTIALIASING,

RenderingHints.VALUE\_ANTIALIAS\_ON);

rh.put(RenderingHints.KEY\_RENDERING,

RenderingHints.VALUE\_RENDER\_QUALITY);

g2d.setRenderingHints(rh);

Dimension size = getSize();

double w = size.getWidth();

double h = size.getHeight();

Ellipse2D e = new Ellipse2D.Double(0, 0, 80, 130);

g2d.setStroke(new BasicStroke(1));

g2d.setColor(Color.gray);

for (double deg = 0; deg < 360; deg += 5) {

AffineTransform at

= AffineTransform.getTranslateInstance(w/2, h/2);

at.rotate(Math.toRadians(deg));

g2d.draw(at.createTransformedShape(e));

}

}

}

The painting is done inside the paintComponent() method.

Graphics2D g2d = (Graphics2D) g;

The Graphics2D class extends the Graphics class. It provides more sophisticated control over geometry, coordinate transformations, colour management, and text layout.

private void drawDonut(Graphics g) {

...

}

It is a good programming practice to delegate the actual painting to a specific method.

RenderingHints rh

= new RenderingHints(RenderingHints.KEY\_ANTIALIASING,

RenderingHints.VALUE\_ANTIALIAS\_ON);

rh.put(RenderingHints.KEY\_RENDERING,

RenderingHints.VALUE\_RENDER\_QUALITY);

g2d.setRenderingHints(rh);

The rendering hints are used to make the drawing smooth.

Dimension size = getSize();

double w = size.getWidth();

double h = size.getHeight();

We get the height and the width of the window. We need them to center the donut shape on the window.

Ellipse2D e = new Ellipse2D.Double(0, 0, 80, 130);

g2d.setStroke(new BasicStroke(1));

g2d.setColor(Color.gray);

Here we create the ellipse.

for (double deg = 0; deg < 360; deg += 5) {

AffineTransform at

= AffineTransform.getTranslateInstance(w/2, h/2);

at.rotate(Math.toRadians(deg));

g2d.draw(at.createTransformedShape(e));

}

Here the ellipse is rotated 72 times to create a donut shape.

Donut.java

package com.zetcode;

import java.awt.EventQueue;

import javax.swing.JFrame;

public class DonutExample extends JFrame {

public DonutExample() {

initUI();

}

private void initUI() {

add(new Board());

setSize(330, 330);

setTitle("Donut");

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setLocationRelativeTo(null);

}

public static void main(String[] args) {

EventQueue.invokeLater(new Runnable() {

@Override

public void run() {

DonutExample ex = new DonutExample();

ex.setVisible(true);

}

});

}

}

This is the main class.

**Drawing an image**

When we create computer games we often work with images. In the next example we load an image and paint it on the board.

Board.java

package com.zetcode;

import java.awt.Dimension;

import java.awt.Graphics;

import java.awt.Image;

import javax.swing.ImageIcon;

import javax.swing.JPanel;

public class Board extends JPanel {

private Image bardejov;

public Board() {

initBoard();

}

private void initBoard() {

loadImage();

int w = bardejov.getWidth(this);

int h = bardejov.getHeight(this);

setPreferredSize(new Dimension(w, h));

}

private void loadImage() {

ImageIcon ii = new ImageIcon("bardejov.png");

bardejov = ii.getImage();

}

@Override

public void paintComponent(Graphics g) {

g.drawImage(bardejov, 0, 0, null);

}

}

We pain an image of a town on the board. The image is drawn inside the paintComponent() method.

ImageIcon ii = new ImageIcon("bardejov.png");

We create an ImageIcon.

bardejov = ii.getImage();

We get an Image out of the ImageIcon.

g.drawImage(bardejov, 0, 0, null);

We draw the image on the window.

int w = bardejov.getWidth(this);

int h = bardejov.getHeight(this);

setPreferredSize(new Dimension(w, h));

We determine the width and height of the image. The preferred size of the board panel is set to the dimensions of the image. In cooperation with the JFrame's pack() method, the window is just big enough to show the image.

Image.java

package com.zetcode;

import java.awt.EventQueue;

import javax.swing.JFrame;

public class ImageExample extends JFrame {

public ImageExample() {

initUI();

}

private void initUI() {

add(new Board());

pack();

setTitle("Bardejov");

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setLocationRelativeTo(null);

}

public static void main(String[] args) {

EventQueue.invokeLater(new Runnable() {

@Override

public void run() {

ImageExample ex = new ImageExample();

ex.setVisible(true);

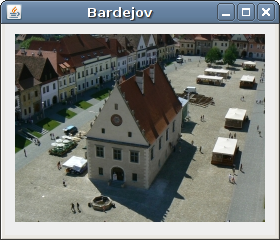
}

});

}

}

This is the main class of the example.

Figure: Image

In this chapter, we have covered some basics of Java game programming.

**Animation**

In this part of the Java 2D games tutorial, we will work with animation.

*Animation* is a rapid display of sequence of images which creates an illusion of movement. We will animate a star on our Board. We will implement the movement in three basic ways. We will use a Swing timer, a standard utility timer, and a thread.

Animation is a complex subject in game programming. Java games are expected to run on multiple operating systems with different hardware specifications. Threads give the most accurate timing solutions. However, for our simple 2D games, other two options can be an option too.

**Swing timer**

In the first example we will use a Swing timer to create animation. This is the easiest but also the least effective way of animating objects in Java games.

SwingTimerExample.java

package com.zetcode;

import java.awt.EventQueue;

import javax.swing.JFrame;

public class SwingTimerExample extends JFrame {

public SwingTimerExample() {

initUI();

}

private void initUI() {

add(new Board());

setResizable(false);

pack();

setTitle("Star");

setLocationRelativeTo(null);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

public static void main(String[] args) {

EventQueue.invokeLater(new Runnable() {

@Override

public void run() {

JFrame ex = new SwingTimerExample();

ex.setVisible(true);

}

});

}

}

This is the main class for the code example.

setResizable(false);

pack();

The setResizable() sets whether the frame can be resized. The pack() method causes this window to be sized to fit the preferred size and layouts of its children. Note that the order in which these two methods are called is important. (The setResizable() changes the insets of the frame on some platforms; calling this method after the pack() method might lead to incorrect results—the star would not go precisely into the right-bottom border of the window.)

Board.java

package com.zetcode;

import java.awt.Color;

import java.awt.Dimension;

import java.awt.Graphics;

import java.awt.Graphics2D;

import java.awt.Image;

import java.awt.Toolkit;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.ImageIcon;

import javax.swing.JPanel;

import javax.swing.Timer;

public class Board extends JPanel

implements ActionListener {

private final int B\_WIDTH = 350;

private final int B\_HEIGHT = 350;

private final int INITIAL\_X = -40;

private final int INITIAL\_Y = -40;

private final int DELAY = 25;

private Image star;

private Timer timer;

private int x, y;

public Board() {

initBoard();

}

private void loadImage() {

ImageIcon ii = new ImageIcon("star.png");

star = ii.getImage();

}

private void initBoard() {

setBackground(Color.BLACK);

setPreferredSize(new Dimension(B\_WIDTH, B\_HEIGHT));

setDoubleBuffered(true);

loadImage();

x = INITIAL\_X;

y = INITIAL\_Y;

timer = new Timer(DELAY, this);

timer.start();

}

@Override

public void paintComponent(Graphics g) {

super.paintComponent(g);

drawStar(g);

}

private void drawStar(Graphics g) {

g.drawImage(star, x, y, this);

Toolkit.getDefaultToolkit().sync();

}

@Override

public void actionPerformed(ActionEvent e) {

x += 1;

y += 1;

if (y > B\_HEIGHT) {

y = INITIAL\_Y;

x = INITIAL\_X;

}

repaint();

}

}

In the Board class we move a star that from the upper-left corner to the right-bottom corner.

private final int B\_WIDTH = 350;

private final int B\_HEIGHT = 350;

private final int INITIAL\_X = -40;

private final int INITIAL\_Y = -40;

private final int DELAY = 25;

Five constants are defined. The first two constants are the board width and height. The third and fourth are the initial coordinates of the star. The last one determines the speed of the animation.

private void loadImage() {

ImageIcon ii = new ImageIcon("star.png");

star = ii.getImage();

}

In the loadImage() method we create an instance of the ImageIcon class. The image is located in the project directory. The getImage() method will return the the Image object from this class. This object will be drawn on the board.

setDoubleBuffered(true);

The JPanel component will use a buffer to paint. This means that all drawing will be done in memory first. Later the off-screen buffer will be copied to the screen. In this simple example, we might not notice any differences.

timer = new Timer(DELAY, this);

timer.start();

Here we create a Swing Timer class and call its start() method. Every DELAY ms the timer will call the actionPerformed() method. In order to use the actionPerformed() method, we must implement the ActionListener interface.

@Override

public void paintComponent(Graphics g) {

super.paintComponent(g);

drawStar(g);

}

Custom painting is done in the paintComponent() method. Note that we also call the paintComponent() method of its parent. The actual painting is delegated to the drawStar() method.

private void drawStar(Graphics g) {

g.drawImage(star, x, y, this);

Toolkit.getDefaultToolkit().sync();

}

In the drawStar() method, we draw the image on the window with the usage of the drawImage() method. The Toolkit.getDefaultToolkit().sync() synchronises the painting on systems that buffer graphics events. Without this line, the animation might not be smooth on Linux.

@Override

public void actionPerformed(ActionEvent e) {

x += 1;

y += 1;

if (y > B\_HEIGHT) {

y = INITIAL\_Y;

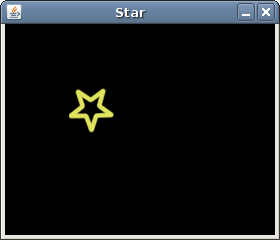
x = INITIAL\_X;

}

repaint();

}

The actionPerformed() method is repeatedly called by the timer. Inside the method, we increase the x and y values of the star object. Then we call the repaint() method which will cause the paintComponent() to be called. This way we regularly repaint the Board thus making the animation.

Figure: Star

**Utility timer**

This is very similar to the previous way. We use the java.util.Timer instead of the javax.Swing.Timer. For Java Swing games this way should be more accurate.

UtilityTimerExample.java

package com.zetcode;

import java.awt.EventQueue;

import javax.swing.JFrame;

public class UtilityTimerExample extends JFrame {

public UtilityTimerExample() {

initUI();

}

private void initUI() {

add(new Board());

setResizable(false);

pack();

setTitle("Star");

setLocationRelativeTo(null);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

public static void main(String[] args) {

EventQueue.invokeLater(new Runnable() {

@Override

public void run() {

JFrame ex = new UtilityTimerExample();

ex.setVisible(true);

}

});

}

}

This is the main class.

Board.java

package com.zetcode;

import java.awt.Color;

import java.awt.Dimension;

import java.awt.Graphics;

import java.awt.Image;

import java.awt.Toolkit;

import java.util.Timer;

import java.util.TimerTask;

import javax.swing.ImageIcon;

import javax.swing.JPanel;

public class Board extends JPanel {

private final int B\_WIDTH = 350;

private final int B\_HEIGHT = 350;

private final int INITIAL\_X = -40;

private final int INITIAL\_Y = -40;

private final int INITIAL\_DELAY = 100;

private final int PERIOD\_INTERVAL = 25;

private Image star;

private Timer timer;

private int x, y;

public Board() {

initBoard();

}

private void loadImage() {

ImageIcon ii = new ImageIcon("star.png");

star = ii.getImage();

}

private void initBoard() {

setBackground(Color.BLACK);

setPreferredSize(new Dimension(B\_WIDTH, B\_HEIGHT));

setDoubleBuffered(true);

loadImage();

x = INITIAL\_X;

y = INITIAL\_Y;

timer = new Timer();

timer.scheduleAtFixedRate(new ScheduleTask(),

INITIAL\_DELAY, PERIOD\_INTERVAL);

}

@Override

public void paintComponent(Graphics g) {

super.paintComponent(g);

drawStar(g);

}

private void drawStar(Graphics g) {

g.drawImage(star, x, y, this);

Toolkit.getDefaultToolkit().sync();

}

private class ScheduleTask extends TimerTask {

@Override

public void run() {

x += 1;

y += 1;

if (y > B\_HEIGHT) {

y = INITIAL\_Y;

x = INITIAL\_X;

}

repaint();

}

}

}

In this example, the timer will regularly call the run() method of the ScheduleTask class.

timer = new Timer();

timer.scheduleAtFixedRate(new ScheduleTask(),

INITIAL\_DELAY, PERIOD\_INTERVAL);

Here we create a timer and schedule a task with a specific interval. There is an initial delay.

@Override

public void run() {

...

}

Each 10ms the timer will call this run() method.

**Thread**

Animating objects using a thread is the most effective and accurate way of animation.

ThreadAnimationExample.java

package com.zetcode;

import java.awt.EventQueue;

import javax.swing.JFrame;

public class ThreadAnimationExample extends JFrame {

public ThreadAnimationExample() {

initUI();

}

private void initUI() {

add(new Board());

setResizable(false);

pack();

setTitle("Star");

setLocationRelativeTo(null);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

public static void main(String[] args) {

EventQueue.invokeLater(new Runnable() {

@Override

public void run() {

JFrame ex = new ThreadAnimationExample();

ex.setVisible(true);

}

});

}

}

This is the main class.

Board.java

package com.zetcode;

import java.awt.Color;

import java.awt.Dimension;

import java.awt.Graphics;

import java.awt.Image;

import java.awt.Toolkit;

import javax.swing.ImageIcon;

import javax.swing.JPanel;

public class Board extends JPanel

implements Runnable {

private final int B\_WIDTH = 350;

private final int B\_HEIGHT = 350;

private final int INITIAL\_X = -40;

private final int INITIAL\_Y = -40;

private final int DELAY = 25;

private Image star;

private Thread animator;

private int x, y;

public Board() {

initBoard();

}

private void loadImage() {

ImageIcon ii = new ImageIcon("star.png");

star = ii.getImage();

}

private void initBoard() {

setBackground(Color.BLACK);

setPreferredSize(new Dimension(B\_WIDTH, B\_HEIGHT));

setDoubleBuffered(true);

loadImage();

x = INITIAL\_X;

y = INITIAL\_Y;

}

@Override

public void addNotify() {

super.addNotify();

animator = new Thread(this);

animator.start();

}

@Override

public void paintComponent(Graphics g) {

super.paintComponent(g);

drawStar(g);

}

private void drawStar(Graphics g) {

g.drawImage(star, x, y, this);

Toolkit.getDefaultToolkit().sync();

}

private void cycle() {

x += 1;

y += 1;

if (y > B\_HEIGHT) {

y = INITIAL\_Y;

x = INITIAL\_X;

}

}

@Override

public void run() {

long beforeTime, timeDiff, sleep;

beforeTime = System.currentTimeMillis();

while (true) {

cycle();

repaint();

timeDiff = System.currentTimeMillis() - beforeTime;

sleep = DELAY - timeDiff;

if (sleep < 0) {

sleep = 2;

}

try {

Thread.sleep(sleep);

} catch (InterruptedException e) {

System.out.println("Interrupted: " + e.getMessage());

}

beforeTime = System.currentTimeMillis();

}

}

}

In the previous examples, we executed a task at specific intervals. In this example, the animation will take place inside a thread. The run() method is called only once. This is why why we have a while loop in the method. From this method, we call the cycle() and the repaint() methods.

@Override

public void addNotify() {

super.addNotify();

animator = new Thread(this);

animator.start();

}

The addNotify() method is called after our JPanel has been added to the JFrame component. This method is often used for various initialisation tasks.

We want our game run smoothly, at constant speed. Therefore we compute the system time.

timeDiff = System.currentTimeMillis() - beforeTime;

sleep = DELAY - timeDiff;

The cycle() and the repaint() methods might take different time at various while cycles. We calculate the time both methods run and subtract it from the DELAY constant. This way we want to ensure that each while cycle runs at constant time. In our case, it is DELAY ms each cycle.

This part of the Java 2D games tutorial covered animation.

**Moving sprites**

In this part of the Java 2D games tutorial we will work with sprites.

The term *sprite* has several meanings. It is used to denote an image or an animation in a scene. It is also used to represent any movable object in a game. Also one of the meanings is the code that encapsulates a character in a game. In our tutorial by using sprite we refer to a movable object or its Java class.

**Moving sprite**

In the first example we have a spacecraft. We can move the spacecraft on the board using the cursor keys.

Craft.java

package com.zetcode;

import java.awt.Image;

import java.awt.event.KeyEvent;

import javax.swing.ImageIcon;

public class Craft {

private int dx;

private int dy;

private int x;

private int y;

private Image image;

public Craft() {

initCraft();

}

private void initCraft() {

ImageIcon ii = new ImageIcon("craft.png");

image = ii.getImage();

x = 40;

y = 60;

}

public void move() {

x += dx;

y += dy;

}

public int getX() {

return x;

}

public int getY() {

return y;

}

public Image getImage() {

return image;

}

public void keyPressed(KeyEvent e) {

int key = e.getKeyCode();

if (key == KeyEvent.VK\_LEFT) {

dx = -1;

}

if (key == KeyEvent.VK\_RIGHT) {

dx = 1;

}

if (key == KeyEvent.VK\_UP) {

dy = -1;

}

if (key == KeyEvent.VK\_DOWN) {

dy = 1;

}

}

public void keyReleased(KeyEvent e) {

int key = e.getKeyCode();

if (key == KeyEvent.VK\_LEFT) {

dx = 0;

}

if (key == KeyEvent.VK\_RIGHT) {

dx = 0;

}

if (key == KeyEvent.VK\_UP) {

dy = 0;

}

if (key == KeyEvent.VK\_DOWN) {

dy = 0;

}

}

}

This class represents a spacecraft. In this class we keep the image of the sprite and the coordinates of the sprite. The keyPressed() and keyReleased() methods control whether the sprite is moving.

public void move() {

x += dx;

y += dy;

}

The move() method changes the coordinates of the sprite. These x and y values are used in the paintComponent() method to draw the image of the sprite.

if (key == KeyEvent.VK\_LEFT) {

dx = 0;

}

When we release the left cursor key, we set the dx variable to zero. The spacecraft will stop moving.

Board.java

package com.zetcode;

import java.awt.Color;

import java.awt.Graphics;

import java.awt.Graphics2D;

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 javax.swing.JPanel;

import javax.swing.Timer;

public class Board extends JPanel implements ActionListener {

private Timer timer;

private Craft craft;

private final int DELAY = 10;

public Board() {

initBoard();

}

private void initBoard() {

addKeyListener(new TAdapter());

setFocusable(true);

setBackground(Color.BLACK);

craft = new Craft();

timer = new Timer(DELAY, this);

timer.start();

}

@Override

public void paintComponent(Graphics g) {

super.paintComponent(g);

doDrawing(g);

Toolkit.getDefaultToolkit().sync();

}

private void doDrawing(Graphics g) {

Graphics2D g2d = (Graphics2D) g;

g2d.drawImage(craft.getImage(), craft.getX(), craft.getY(), this);

}

@Override

public void actionPerformed(ActionEvent e) {

craft.move();

repaint();

}

private class TAdapter extends KeyAdapter {

@Override

public void keyReleased(KeyEvent e) {

craft.keyReleased(e);

}

@Override

public void keyPressed(KeyEvent e) {

craft.keyPressed(e);

}

}

}

This is the Board class.

private void doDrawing(Graphics g) {

Graphics2D g2d = (Graphics2D) g;

g2d.drawImage(craft.getImage(), craft.getX(), craft.getY(), this);

}

In the doDrawing() method, we draw the spacecraft with the drawImage() method. We get the image and the coordinates from the sprite class.

@Override

public void actionPerformed(ActionEvent e) {

craft.move();

repaint();

}

The actionPerformed() method is called every DELAY ms. We move the sprite and repaint the board.

private class TAdapter extends KeyAdapter {

@Override

public void keyReleased(KeyEvent e) {

craft.keyReleased(e);

}

@Override

public void keyPressed(KeyEvent e) {

craft.keyPressed(e);

}

}

In the Board class we listen for key events. The overridden methods of the KeyAdapter class delegate the processing to the methods of the Craft class.

MovingSpriteEx.java

package com.zetcode;

import java.awt.EventQueue;

import javax.swing.JFrame;

public class MovingSpriteEx extends JFrame {

public MovingSpriteEx() {

initUI();

}

private void initUI() {

add(new Board());

setSize(400, 300);

setResizable(false);

setTitle("Moving sprite");

setLocationRelativeTo(null);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

public static void main(String[] args) {

EventQueue.invokeLater(new Runnable() {

@Override

public void run() {

MovingSpriteEx ex = new MovingSpriteEx();

ex.setVisible(true);

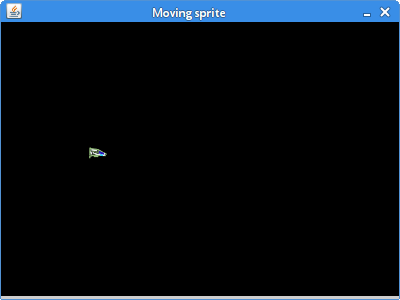
}

});

}

}

This is the main class.

Figure: Moving sprite

**Shooting missiles**

In the next example we will add another sprite type to our example—a missile. The missiles are launched with the Space key.

Sprite.java

package com.zetcode;

import java.awt.Image;

import javax.swing.ImageIcon;

public class Sprite {

protected int x;

protected int y;

protected int width;

protected int height;

protected boolean vis;

protected Image image;

public Sprite(int x, int y) {

this.x = x;

this.y = y;

vis = true;

}

protected void loadImage(String imageName) {

ImageIcon ii = new ImageIcon(imageName);

image = ii.getImage();

}

protected void getImageDimensions() {

width = image.getWidth(null);

height = image.getHeight(null);

}

public Image getImage() {

return image;

}

public int getX() {

return x;

}

public int getY() {

return y;

}

public boolean isVisible() {

return vis;

}

public void setVisible(Boolean visible) {

vis = visible;

}

}

The Sprite class shares common code from the Missile and Craft classes.

public Sprite(int x, int y) {

this.x = x;

this.y = y;

vis = true;

}

The constructor initiates the x and y coordiates and the vis variable.

Missile.java

package com.zetcode;

public class Missile extends Sprite {

private final int BOARD\_WIDTH = 390;

private final int MISSILE\_SPEED = 2;

public Missile(int x, int y) {

super(x, y);

initMissile();

}

private void initMissile() {

loadImage("missile.png");

getImageDimensions();

}

public void move() {

x += MISSILE\_SPEED;

if (x > BOARD\_WIDTH) {

vis = false;

}

}

}

Here we have a new sprite called Missile.

public void move() {

x += MISSILE\_SPEED;

if (x > BOARD\_WIDTH) {

vis = false;

}

}

The missile moves at constant speed. When it hits the right border of the Board, it becomes invisible. It is then removed from the list of missiles.

Craft.java

package com.zetcode;

import java.awt.event.KeyEvent;

import java.util.ArrayList;

public class Craft extends Sprite {

private int dx;

private int dy;

private ArrayList missiles;

public Craft(int x, int y) {

super(x, y);

initCraft();

}

private void initCraft() {

missiles = new ArrayList();

loadImage("craft.png");

getImageDimensions();

}

public void move() {

x += dx;

y += dy;

}

public ArrayList getMissiles() {

return missiles;

}

public void keyPressed(KeyEvent e) {

int key = e.getKeyCode();

if (key == KeyEvent.VK\_SPACE) {

fire();

}

if (key == KeyEvent.VK\_LEFT) {

dx = -1;

}

if (key == KeyEvent.VK\_RIGHT) {

dx = 1;

}

if (key == KeyEvent.VK\_UP) {

dy = -1;

}

if (key == KeyEvent.VK\_DOWN) {

dy = 1;

}

}

public void fire() {

missiles.add(new Missile(x + width, y + height / 2));

}

public void keyReleased(KeyEvent e) {

int key = e.getKeyCode();

if (key == KeyEvent.VK\_LEFT) {

dx = 0;

}

if (key == KeyEvent.VK\_RIGHT) {

dx = 0;

}

if (key == KeyEvent.VK\_UP) {

dy = 0;

}

if (key == KeyEvent.VK\_DOWN) {

dy = 0;

}

}

}

This is the Craft class.

if (key == KeyEvent.VK\_SPACE) {

fire();

}

If we press the Space key, we fire.

public void fire() {

missiles.add(new Missile(x + width, y + height / 2));

}

The fire() method creates a new Missile object and adds it to the missiles ArrayList.

public ArrayList getMissiles() {

return missiles;

}

The getMissiles() method returns the ArrayList of missiles. It is called from the Board class.

Board.java

package com.zetcode;

import java.awt.Color;

import java.awt.Graphics;

import java.awt.Graphics2D;

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 javax.swing.JPanel;

import javax.swing.Timer;

public class Board extends JPanel implements ActionListener {

private final int ICRAFT\_X = 40;

private final int ICRAFT\_Y = 60;

private final int DELAY = 10;

private Timer timer;

private Craft craft;

public Board() {

initBoard();

}

private void initBoard() {

addKeyListener(new TAdapter());

setFocusable(true);

setBackground(Color.BLACK);

setDoubleBuffered(true);

craft = new Craft(ICRAFT\_X, ICRAFT\_Y);

timer = new Timer(DELAY, this);

timer.start();

}

@Override

public void paintComponent(Graphics g) {

super.paintComponent(g);

doDrawing(g);

Toolkit.getDefaultToolkit().sync();

}

private void doDrawing(Graphics g) {

Graphics2D g2d = (Graphics2D) g;

g2d.drawImage(craft.getImage(), craft.getX(),

craft.getY(), this);

ArrayList ms = craft.getMissiles();

for (Object m1 : ms) {

Missile m = (Missile) m1;

g2d.drawImage(m.getImage(), m.getX(),

m.getY(), this);

}

}

@Override

public void actionPerformed(ActionEvent e) {

updateMissiles();

updateCraft();

repaint();

}

private void updateMissiles() {

ArrayList ms = craft.getMissiles();

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

Missile m = (Missile) ms.get(i);

if (m.isVisible()) {

m.move();

} else {

ms.remove(i);

}

}

}

private void updateCraft() {

craft.move();

}

private class TAdapter extends KeyAdapter {

@Override

public void keyReleased(KeyEvent e) {

craft.keyReleased(e);

}

@Override

public void keyPressed(KeyEvent e) {

craft.keyPressed(e);

}

}

}

This is the Board class.

private void doDrawing(Graphics g) {

Graphics2D g2d = (Graphics2D) g;

g2d.drawImage(craft.getImage(), craft.getX(),

craft.getY(), this);

ArrayList ms = craft.getMissiles();

for (Object m1 : ms) {

Missile m = (Missile) m1;

g2d.drawImage(m.getImage(), m.getX(),

m.getY(), this);

}

}

In the doDrawing() method, we draw the craft and all the available missiles.

private void updateMissiles() {

ArrayList ms = craft.getMissiles();

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

Missile m = (Missile) ms.get(i);

if (m.isVisible()) {

m.move();

} else {

ms.remove(i);

}

}

}

In the updateMissiles() method we parse all missiles from the missiles list. Depending on what the isVisible() method returns, we either move the missile or remove it from the container.

ShootingMissilesEx.java

package com.zetcode;

import java.awt.EventQueue;

import javax.swing.JFrame;

public class ShootingMissilesEx extends JFrame {

public ShootingMissilesEx() {

initUI();

}

private void initUI() {

add(new Board());

setSize(400, 300);

setResizable(false);

setTitle("Shooting missiles");

setLocationRelativeTo(null);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

public static void main(String[] args) {

EventQueue.invokeLater(new Runnable() {

@Override

public void run() {

ShootingMissilesEx ex = new ShootingMissilesEx();

ex.setVisible(true);

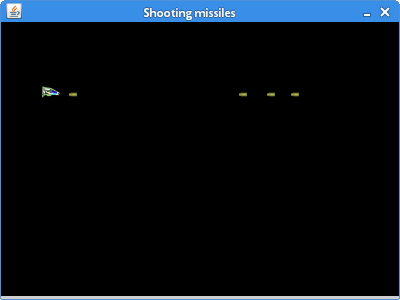
}

});

}

}

Finally, this is the main class.

Figure: Shooting missiles

In this chapter, we have covered sprites.

**Collision detection**

In this part of the Java 2D games tutorial we will talk about collision detection.

Many games need to handle collisions, especially arcade games. Simply said, we need to detect when two objects collide on screen. In the next code example, we will expand the previous example. We add a new Alien sprite. We will detect two types collisions: when the missile hits an alien ship and when our spacecraft collides with an alien.

**Shooting aliens**

In the example, we have a spacecraft and aliens. We can move the spacecraft on the board using the cursor keys. Missiles destroying aliens are launched with the spacebar key.

Sprite.java

package com.zetcode;

import java.awt.Image;

import java.awt.Rectangle;

import javax.swing.ImageIcon;

public class Sprite {

protected int x;

protected int y;

protected int width;

protected int height;

protected boolean vis;

protected Image image;

public Sprite(int x, int y) {

this.x = x;

this.y = y;

vis = true;

}

protected void getImageDimensions() {

width = image.getWidth(null);

height = image.getHeight(null);

}

protected void loadImage(String imageName) {

ImageIcon ii = new ImageIcon(imageName);

image = ii.getImage();

}

public Image getImage() {

return image;

}

public int getX() {

return x;

}

public int getY() {

return y;

}

public boolean isVisible() {

return vis;

}

public void setVisible(Boolean visible) {

vis = visible;

}

public Rectangle getBounds() {

return new Rectangle(x, y, width, height);

}

}

The code that can be shared by all sprites (a craft, an alien, and a missile) is placed in the Sprite class.

public Rectangle getBounds() {

return new Rectangle(x, y, width, height);

}

The getBounds() method returns the bounding rectangle of the sprite image. We need the bounds in collision detection.

Craft.java

package com.zetcode;

import java.awt.event.KeyEvent;

import java.util.ArrayList;

public class Craft extends Sprite {

private int dx;

private int dy;

private ArrayList<Missile> missiles;

public Craft(int x, int y) {

super(x, y);

initCraft();

}

private void initCraft() {

missiles = new ArrayList<>();

loadImage("craft.png");

getImageDimensions();

}

public void move() {

x += dx;

y += dy;

if (x < 1) {

x = 1;

}

if (y < 1) {

y = 1;

}

}

public ArrayList getMissiles() {

return missiles;

}

public void keyPressed(KeyEvent e) {

int key = e.getKeyCode();

if (key == KeyEvent.VK\_SPACE) {

fire();

}

if (key == KeyEvent.VK\_LEFT) {

dx = -1;

}

if (key == KeyEvent.VK\_RIGHT) {

dx = 1;

}

if (key == KeyEvent.VK\_UP) {

dy = -1;

}

if (key == KeyEvent.VK\_DOWN) {

dy = 1;

}

}

public void fire() {

missiles.add(new Missile(x + width, y + height / 2));

}

public void keyReleased(KeyEvent e) {

int key = e.getKeyCode();

if (key == KeyEvent.VK\_LEFT) {

dx = 0;

}

if (key == KeyEvent.VK\_RIGHT) {

dx = 0;

}

if (key == KeyEvent.VK\_UP) {

dy = 0;

}

if (key == KeyEvent.VK\_DOWN) {

dy = 0;

}

}

}

This class represents a spacecraft.

private ArrayList<Missile> missiles;

All the missiles fired by the spacecraft are stored in the missiles list.

public void fire() {

missiles.add(new Missile(x + width, y + height / 2));

}

When we fire a missile, a new Missile object is added to the missiles list. It is retained in the list until it collides with an alien or goes out of the window.

Board.java

package com.zetcode;

import java.awt.Color;

import java.awt.Dimension;

import java.awt.Font;

import java.awt.FontMetrics;

import java.awt.Graphics;

import java.awt.Rectangle;

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 javax.swing.JPanel;

import javax.swing.Timer;

public class Board extends JPanel implements ActionListener {

private Timer timer;

private Craft craft;

private ArrayList<Alien> aliens;

private boolean ingame;

private final int ICRAFT\_X = 40;

private final int ICRAFT\_Y = 60;

private final int B\_WIDTH = 400;

private final int B\_HEIGHT = 300;

private final int DELAY = 15;

private final int[][] pos = {

{2380, 29}, {2500, 59}, {1380, 89},

{780, 109}, {580, 139}, {680, 239},

{790, 259}, {760, 50}, {790, 150},

{980, 209}, {560, 45}, {510, 70},

{930, 159}, {590, 80}, {530, 60},

{940, 59}, {990, 30}, {920, 200},

{900, 259}, {660, 50}, {540, 90},

{810, 220}, {860, 20}, {740, 180},

{820, 128}, {490, 170}, {700, 30}

};

public Board() {

initBoard();

}

private void initBoard() {

addKeyListener(new TAdapter());

setFocusable(true);

setBackground(Color.BLACK);

ingame = true;

setPreferredSize(new Dimension(B\_WIDTH, B\_HEIGHT));

craft = new Craft(ICRAFT\_X, ICRAFT\_Y);

initAliens();

timer = new Timer(DELAY, this);

timer.start();

}

public void initAliens() {

aliens = new ArrayList<>();

for (int[] p : pos) {

aliens.add(new Alien(p[0], p[1]));

}

}

@Override

public void paintComponent(Graphics g) {

super.paintComponent(g);

if (ingame) {

drawObjects(g);

} else {

drawGameOver(g);

}

Toolkit.getDefaultToolkit().sync();

}

private void drawObjects(Graphics g) {

if (craft.isVisible()) {

g.drawImage(craft.getImage(), craft.getX(), craft.getY(),

this);

}

ArrayList<Missile> ms = craft.getMissiles();

for (Missile m : ms) {

if (m.isVisible()) {

g.drawImage(m.getImage(), m.getX(), m.getY(), this);

}

}

for (Alien a : aliens) {

if (a.isVisible()) {

g.drawImage(a.getImage(), a.getX(), a.getY(), this);

}

}

g.setColor(Color.WHITE);

g.drawString("Aliens left: " + aliens.size(), 5, 15);

}

private void drawGameOver(Graphics g) {

String msg = "Game Over";

Font small = new Font("Helvetica", Font.BOLD, 14);

FontMetrics fm = getFontMetrics(small);

g.setColor(Color.white);

g.setFont(small);

g.drawString(msg, (B\_WIDTH - fm.stringWidth(msg)) / 2,

B\_HEIGHT / 2);

}

@Override

public void actionPerformed(ActionEvent e) {

inGame();

updateCraft();

updateMissiles();

updateAliens();

checkCollisions();

repaint();

}

private void inGame() {

if (!ingame) {

timer.stop();

}

}

private void updateCraft() {

if (craft.isVisible()) {

craft.move();

}

}

private void updateMissiles() {

ArrayList<Missile> ms = craft.getMissiles();

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

Missile m = ms.get(i);

if (m.isVisible()) {

m.move();

} else {

ms.remove(i);

}

}

}

private void updateAliens() {

if (aliens.isEmpty()) {

ingame = false;

return;

}

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

Alien a = aliens.get(i);

if (a.isVisible()) {

a.move();

} else {

aliens.remove(i);

}

}

}

public void checkCollisions() {

Rectangle r3 = craft.getBounds();

for (Alien alien : aliens) {

Rectangle r2 = alien.getBounds();

if (r3.intersects(r2)) {

craft.setVisible(false);

alien.setVisible(false);

ingame = false;

}

}

ArrayList<Missile> ms = craft.getMissiles();

for (Missile m : ms) {

Rectangle r1 = m.getBounds();

for (Alien alien : aliens) {

Rectangle r2 = alien.getBounds();

if (r1.intersects(r2)) {

m.setVisible(false);

alien.setVisible(false);

}

}

}

}

private class TAdapter extends KeyAdapter {

@Override

public void keyReleased(KeyEvent e) {

craft.keyReleased(e);

}

@Override

public void keyPressed(KeyEvent e) {

craft.keyPressed(e);

}

}

}

This is the Board class.

private final int[][] pos = {

{2380, 29}, {2500, 59}, {1380, 89},

{780, 109}, {580, 139}, {680, 239},

{790, 259}, {760, 50}, {790, 150},

{980, 209}, {560, 45}, {510, 70},

{930, 159}, {590, 80}, {530, 60},

{940, 59}, {990, 30}, {920, 200},

{900, 259}, {660, 50}, {540, 90},

{810, 220}, {860, 20}, {740, 180},

{820, 128}, {490, 170}, {700, 30}

};

These are the initial positions of alien ships.

public void initAliens() {

aliens = new ArrayList<>();

for (int[] p : pos) {

aliens.add(new Alien(p[0], p[1]));

}

}

The initAliens() method creates a list of alien objects. The aliens take their initial positions from the pos array.

@Override

public void paintComponent(Graphics g) {

super.paintComponent(g);

if (ingame) {

drawObjects(g);

} else {

drawGameOver(g);

}

Toolkit.getDefaultToolkit().sync();

}

Inside the paintComponent() method, we either draw game sprites or write the game over message. This depends on the ingame variable.

private void drawObjects(Graphics g) {

if (craft.isVisible()) {

g.drawImage(craft.getImage(), craft.getX(), craft.getY(),

this);

}

...

}

The drawObjects() method draws game sprites on the window. First, we draw the craft sprite.

for (Alien a : aliens) {

if (a.isVisible()) {

g.drawImage(a.getImage(), a.getX(), a.getY(), this);

}

}

In this loop we draw all aliens; they are drawn only if they have not been previously destroyed. This is checked by the isVisible() method.

g.setColor(Color.WHITE);

g.drawString("Aliens left: " + aliens.size(), 5, 15);

In the top-left corner of the window, we draw how many aliens are left.

private void drawGameOver(Graphics g) {

String msg = "Game Over";

Font small = new Font("Helvetica", Font.BOLD, 14);

FontMetrics fm = getFontMetrics(small);

g.setColor(Color.white);

g.setFont(small);

g.drawString(msg, (B\_WIDTH - fm.stringWidth(msg)) / 2,

B\_HEIGHT / 2);

}

The drawGameOver() draws a game over message in the middle of the window. The message is displayed at the end of the game, either when we destroy all alien ships or when we collide with one of them.

@Override

public void actionPerformed(ActionEvent e) {

inGame();

updateCraft();

updateMissiles();

updateAliens();

checkCollisions();

repaint();

}

Each action event represents one game cycle. The game logic is factored into specific methods. For instance, the updateMissiles() moves all the available missiles.

private void updateAliens() {

if (aliens.isEmpty()) {

ingame = false;

return;

}

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

Alien a = aliens.get(i);

if (a.isVisible()) {

a.move();

} else {

aliens.remove(i);

}

}

}

Inside the updateAliens() method, we first check if there are any alien objects left in the aliens list. The game is finished if the list is empty. If it is not empty, we go trough the list and move all its items. The destroyed aliens are removed from the list.

public void checkCollisions() {

Rectangle r3 = craft.getBounds();

for (Alien alien : aliens) {

Rectangle r2 = alien.getBounds();

if (r3.intersects(r2)) {

craft.setVisible(false);

alien.setVisible(false);

ingame = false;

}

}

...

}

The checkCollisions() method checks for possible collisions. First, we check if the craft object collides with any of the alien objects. We get the rectangles of the objects with the getBounds() method. The intersects() method checks if the two rectangles intersect.

ArrayList<Missile> ms = craft.getMissiles();

for (Missile m : ms) {

Rectangle r1 = m.getBounds();

for (Alien alien : aliens) {

Rectangle r2 = alien.getBounds();

if (r1.intersects(r2)) {

m.setVisible(false);

alien.setVisible(false);

}

}

}

This code checks the collisions between missiles and aliens.

Alien.java

package com.zetcode;

public class Alien extends Sprite {

private final int INITIAL\_X = 400;

public Alien(int x, int y) {

super(x, y);

initAlien();

}

private void initAlien() {

loadImage("alien.png");

getImageDimensions();

}

public void move() {

if (x < 0) {

x = INITIAL\_X;

}

x -= 1;

}

}

This is the Alien class.

public void move() {

if (x < 0) {

x = INITIAL\_X;

}

x -= 1;

}

Aliens return to the screen on the right side after they have disappeared on the left.

Missile.java

package com.zetcode;

public class Missile extends Sprite {

private final int BOARD\_WIDTH = 390;

private final int MISSILE\_SPEED = 2;

public Missile(int x, int y) {

super(x, y);

initMissile();

}

private void initMissile() {

loadImage("missile.png");

getImageDimensions();

}

public void move() {

x += MISSILE\_SPEED;

if (x > BOARD\_WIDTH)

vis = false;

}

}

This is the Missile class.

public void move() {

x += MISSILE\_SPEED;

if (x > BOARD\_WIDTH)

vis = false;

}

Missiles move in one direction only. They disappear after they reach the right window border.

CollisionEx.java

package com.zetcode;

import java.awt.EventQueue;

import javax.swing.JFrame;

public class CollisionEx extends JFrame {

public CollisionEx() {

initUI();

}

private void initUI() {

add(new Board());

setResizable(false);

pack();

setTitle("Collision");

setLocationRelativeTo(null);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

public static void main(String[] args) {

EventQueue.invokeLater(new Runnable() {

@Override

public void run() {

CollisionEx ex = new CollisionEx();

ex.setVisible(true);

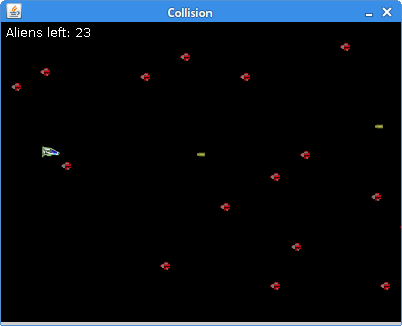
}

});

}

}

Finally, this is the main class.

Figure: Shooting aliens

This chapter was about collision detection

**The Puzzle game**

In this chapter, we will create a simple puzzle game.

**Puzzle**

The goal of this little game is to form a picture. Buttons containing images are moved by clicking on them. Only buttons adjacent to the empty button can be moved.

package com.zetcode;

import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.EventQueue;

import java.awt.Graphics2D;

import java.awt.GridLayout;

import java.awt.Image;

import java.awt.Point;

import java.awt.event.ActionEvent;

import java.awt.event.MouseAdapter;

import java.awt.event.MouseEvent;

import java.awt.image.BufferedImage;

import java.awt.image.CropImageFilter;

import java.awt.image.FilteredImageSource;

import java.io.File;

import java.io.IOException;

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

import java.util.logging.Level;

import java.util.logging.Logger;

import javax.imageio.ImageIO;

import javax.swing.AbstractAction;

import javax.swing.BorderFactory;

import javax.swing.ImageIcon;

import javax.swing.JButton;

import javax.swing.JComponent;

import javax.swing.JFrame;

import javax.swing.JOptionPane;

import javax.swing.JPanel;

class MyButton extends JButton {

private boolean isLastButton;

public MyButton() {

super();

initUI();

}

public MyButton(Image image) {

super(new ImageIcon(image));

initUI();

}

private void initUI() {

isLastButton = false;

BorderFactory.createLineBorder(Color.gray);

addMouseListener(new MouseAdapter() {

@Override

public void mouseEntered(MouseEvent e) {

setBorder(BorderFactory.createLineBorder(Color.yellow));

}

@Override

public void mouseExited(MouseEvent e) {

setBorder(BorderFactory.createLineBorder(Color.gray));

}

});

}

public void setLastButton() {

isLastButton = true;

}

public boolean isLastButton() {

return isLastButton;

}

}

public class PuzzleEx extends JFrame {

private JPanel panel;

private BufferedImage source;

private ArrayList<MyButton> buttons;

ArrayList<Point> solution = new ArrayList();

private Image image;

private MyButton lastButton;

private int width, height;

private final int DESIRED\_WIDTH = 300;

private BufferedImage resized;

public PuzzleEx() {

initUI();

}

private void initUI() {

solution.add(new Point(0, 0));

solution.add(new Point(0, 1));

solution.add(new Point(0, 2));

solution.add(new Point(1, 0));

solution.add(new Point(1, 1));

solution.add(new Point(1, 2));

solution.add(new Point(2, 0));

solution.add(new Point(2, 1));

solution.add(new Point(2, 2));

solution.add(new Point(3, 0));

solution.add(new Point(3, 1));

solution.add(new Point(3, 2));

buttons = new ArrayList();

panel = new JPanel();

panel.setBorder(BorderFactory.createLineBorder(Color.gray));

panel.setLayout(new GridLayout(4, 3, 0, 0));

try {

source = loadImage();

int h = getNewHeight(source.getWidth(), source.getHeight());

resized = resizeImage(source, DESIRED\_WIDTH, h,

BufferedImage.TYPE\_INT\_ARGB);

} catch (IOException ex) {

Logger.getLogger(PuzzleEx.class.getName()).log(

Level.SEVERE, null, ex);

}

width = resized.getWidth(null);

height = resized.getHeight(null);

add(panel, BorderLayout.CENTER);

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

for (int j = 0; j < 3; j++) {

image = createImage(new FilteredImageSource(resized.getSource(),

new CropImageFilter(j \* width / 3, i \* height / 4,

(width / 3), height / 4)));

MyButton button = new MyButton(image);

button.putClientProperty("position", new Point(i, j));

if (i == 3 && j == 2) {

lastButton = new MyButton();

lastButton.setBorderPainted(false);

lastButton.setContentAreaFilled(false);

lastButton.setLastButton();

lastButton.putClientProperty("position", new Point(i, j));

} else {

buttons.add(button);

}

}

}

Collections.shuffle(buttons);

buttons.add(lastButton);

for (int i = 0; i < 12; i++) {

MyButton btn = buttons.get(i);

panel.add(btn);

btn.setBorder(BorderFactory.createLineBorder(Color.gray));

btn.addActionListener(new ClickAction());

}

pack();

setTitle("Puzzle");

setResizable(false);

setLocationRelativeTo(null);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

private int getNewHeight(int w, int h) {

double ratio = DESIRED\_WIDTH / (double) w;

int newHeight = (int) (h \* ratio);

return newHeight;

}

private BufferedImage loadImage() throws IOException {

BufferedImage bimg = ImageIO.read(new File("icesid.jpg"));

return bimg;

}

private BufferedImage resizeImage(BufferedImage originalImage, int width,

int height, int type) throws IOException {

BufferedImage resizedImage = new BufferedImage(width, height, type);

Graphics2D g = resizedImage.createGraphics();

g.drawImage(originalImage, 0, 0, width, height, null);

g.dispose();

return resizedImage;

}

private class ClickAction extends AbstractAction {

@Override

public void actionPerformed(ActionEvent e) {

checkButton(e);

checkSolution();

}

private void checkButton(ActionEvent e) {

int lidx = 0;

for (MyButton button : buttons) {

if (button.isLastButton()) {

lidx = buttons.indexOf(button);

}

}

JButton button = (JButton) e.getSource();

int bidx = buttons.indexOf(button);

if ((bidx - 1 == lidx) || (bidx + 1 == lidx)

|| (bidx - 3 == lidx) || (bidx + 3 == lidx)) {

Collections.swap(buttons, bidx, lidx);

updateButtons();

}

}

private void updateButtons() {

panel.removeAll();

for (JComponent btn : buttons) {

panel.add(btn);

}

panel.validate();

}

}

private void checkSolution() {

ArrayList<Point> current = new ArrayList();

for (JComponent btn : buttons) {

current.add((Point) btn.getClientProperty("position"));

}

if (compareList(solution, current)) {

JOptionPane.showMessageDialog(panel, "Finished",

"Congratulation", JOptionPane.INFORMATION\_MESSAGE);

}

}

public static boolean compareList(List ls1, List ls2) {

return ls1.toString().contentEquals(ls2.toString());

}

public static void main(String[] args) {

EventQueue.invokeLater(new Runnable() {

@Override

public void run() {

PuzzleEx puzzle = new PuzzleEx();

puzzle.setVisible(true);

}

});

}

}

We use an image of a Sid character from the Ice Age movie. We scale the image and cut it into 12 pieces. These pieces are used by JButton components. The last piece is not used; we have an empty button instead. You can download some reasonably large picture and use it in this game.

addMouseListener(new MouseAdapter() {

@Override

public void mouseEntered(MouseEvent e) {

setBorder(BorderFactory.createLineBorder(Color.yellow));

}

@Override

public void mouseExited(MouseEvent e) {

setBorder(BorderFactory.createLineBorder(Color.gray));

}

});

When we hover a mouse pointer over the button, its border changes to yellow colour.

public boolean isLastButton() {

return isLastButton;

}

There is one button that we call the last button. It is a button that does not have an image. Other buttons swap space with this one.

private final int DESIRED\_WIDTH = 300;

The image that we use to form is scaled to have the desired width. With the getNewHeight() method we calculate the new height, keeping the image's ratio.

solution.add(new Point(0, 0));

solution.add(new Point(0, 1));

solution.add(new Point(0, 2));

solution.add(new Point(1, 0));

...

The solution array list stores the correct order of buttons which forms the image. Each button is identified by one Point.

panel.setLayout(new GridLayout(4, 3, 0, 0));

We use a GridLayout to store our components. The layout consists of 4 rows and 3 columns.

image = createImage(new FilteredImageSource(resized.getSource(),

new CropImageFilter(j \* width / 3, i \* height / 4,

(width / 3), height / 4)));

CropImageFilter is used to cut a rectangular shape from the already resized image source. It is meant to be used in conjunction with a FilteredImageSource object to produce cropped versions of existing images.

button.putClientProperty("position", new Point(i, j));

Buttons are identified by their position client property. It is a point containing the button's correct row and colum position in the picture. These properties are used to find out if we have the correct order of buttons in the window.

if (i == 3 && j == 2) {

lastButton = new MyButton();

lastButton.setBorderPainted(false);

lastButton.setContentAreaFilled(false);

lastButton.setLastButton();

lastButton.putClientProperty("position", new Point(i, j));

} else {

buttons.add(button);

}

The button with no image is called the last button; it is placed at the end of the grid in the bottom-right corner. It is the button that swaps its position with the adjacent button that is being clicked. We set its isLastButton flag with the setLastButton() method.

Collections.shuffle(buttons);

buttons.add(lastButton);

We randomly reorder the elements of the buttons list. The last button, i.e. the button with no image, is inserted at the end of the list. It is not supposed to be shuffled, it always goes at the end when we start the Puzzle game.

for (int i = 0; i < 12; i++) {

MyButton btn = buttons.get(i);

panel.add(btn);

btn.setBorder(BorderFactory.createLineBorder(Color.gray));

btn.addActionListener(new ClickAction());

}

All the components from the buttons list are placed on the panel. We create some gray border around the buttons and add a click action listener.

private int getNewHeight(int w, int h) {

double ratio = DESIRED\_WIDTH / (double) w;

int newHeight = (int) (h \* ratio);

return newHeight;

}

The getNewHeight() method calculates the height of the image based on the desired width. The image's ratio is kept. We scale the image using these values.

private BufferedImage loadImage() throws IOException {

BufferedImage bimg = ImageIO.read(new File("icesid.jpg"));

return bimg;

}

A JPG image is loaded from the disk. ImageIO's read() method returns a BufferedImage, which is Swing's important class for manipulating images.

private BufferedImage resizeImage(BufferedImage originalImage, int width,

int height, int type) throws IOException {

BufferedImage resizedImage = new BufferedImage(width, height, type);

Graphics2D g = resizedImage.createGraphics();

g.drawImage(originalImage, 0, 0, width, height, null);

g.dispose();

return resizedImage;

}

The original image is resized by creating a new BufferedImage with new dimensions. We paint from the original image into this new buffered image.

private void checkButton(ActionEvent e) {

int lidx = 0;

for (MyButton button : buttons) {

if (button.isLastButton()) {

lidx = buttons.indexOf(button);

}

}

JButton button = (JButton) e.getSource();

int bidx = buttons.indexOf(button);

if ((bidx - 1 == lidx) || (bidx + 1 == lidx)

|| (bidx - 3 == lidx) || (bidx + 3 == lidx)) {

Collections.swap(buttons, bidx, lidx);

updateButtons();

}

}

Buttons are stored in an array list. This list is then mapped to the grid of the panel. We get the indexes of the last button and the clicked button. They are swapped using the Collections.swap() if they are adjacent.

private void updateButtons() {

panel.removeAll();

for (JComponent btn : buttons) {

panel.add(btn);

}

panel.validate();

}

The updateButtons() method maps the list to the panel's grid. First, all components are removed with the removeAll() method. A for loop is used to go trough the buttons list to add the reordered buttons back to the panel's layout manager. Finally, the validate() method implements the new layout.

private void checkSolution() {

ArrayList<Point> current = new ArrayList();

for (JComponent btn : buttons) {

current.add((Point) btn.getClientProperty("position"));

}

if (compareList(solution, current)) {

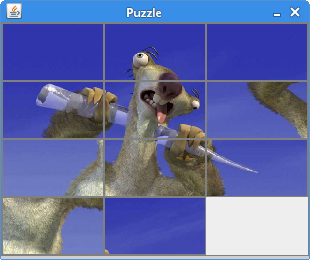
JOptionPane.showMessageDialog(panel, "Finished",

"Congratulation", JOptionPane.INFORMATION\_MESSAGE);

}

}

Solution checking is done by comparing the list of points of the correctly ordered buttons with the current list containg the order of buttons from the window. A message dialog is shown in case the solution is reached.

Figure: Forming the image

This was a puzzle game.

**Snake**

In this part of the Java 2D games tutorial, we will create a Java Snake game clone.

**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*.

**Development**

The size of each of the joints of a snake is 10px. 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.

Board.java

package com.zetcode;

import java.awt.Color;

import java.awt.Dimension;

import java.awt.Font;

import java.awt.FontMetrics;

import java.awt.Graphics;

import java.awt.Image;

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 javax.swing.ImageIcon;

import javax.swing.JPanel;

import javax.swing.Timer;

public class Board extends JPanel implements ActionListener {

private final int B\_WIDTH = 300;

private final int B\_HEIGHT = 300;

private final int DOT\_SIZE = 10;

private final int ALL\_DOTS = 900;

private final int RAND\_POS = 29;

private final int DELAY = 140;

private final int x[] = new int[ALL\_DOTS];

private final int y[] = new int[ALL\_DOTS];

private int dots;

private int apple\_x;

private int apple\_y;

private boolean leftDirection = false;

private boolean rightDirection = true;

private boolean upDirection = false;

private boolean downDirection = false;

private boolean inGame = true;

private Timer timer;

private Image ball;

private Image apple;

private Image head;

public Board() {

addKeyListener(new TAdapter());

setBackground(Color.black);

setFocusable(true);

setPreferredSize(new Dimension(B\_WIDTH, B\_HEIGHT));

loadImages();

initGame();

}

private void loadImages() {

ImageIcon iid = new ImageIcon("dot.png");

ball = iid.getImage();

ImageIcon iia = new ImageIcon("apple.png");

apple = iia.getImage();

ImageIcon iih = new ImageIcon("head.png");

head = iih.getImage();

}

private void initGame() {

dots = 3;

for (int z = 0; z < dots; z++) {

x[z] = 50 - z \* 10;

y[z] = 50;

}

locateApple();

timer = new Timer(DELAY, this);

timer.start();

}

@Override

public void paintComponent(Graphics g) {

super.paintComponent(g);

doDrawing(g);

}

private void doDrawing(Graphics g) {

if (inGame) {

g.drawImage(apple, apple\_x, apple\_y, this);

for (int z = 0; z < dots; z++) {

if (z == 0) {

g.drawImage(head, x[z], y[z], this);

} else {

g.drawImage(ball, x[z], y[z], this);

}

}

Toolkit.getDefaultToolkit().sync();

} else {

gameOver(g);

}

}

private void gameOver(Graphics g) {

String msg = "Game Over";

Font small = new Font("Helvetica", Font.BOLD, 14);

FontMetrics metr = getFontMetrics(small);

g.setColor(Color.white);

g.setFont(small);

g.drawString(msg, (B\_WIDTH - metr.stringWidth(msg)) / 2, B\_HEIGHT / 2);

}

private void checkApple() {

if ((x[0] == apple\_x) && (y[0] == apple\_y)) {

dots++;

locateApple();

}

}

private void move() {

for (int z = dots; z > 0; z--) {

x[z] = x[(z - 1)];

y[z] = y[(z - 1)];

}

if (leftDirection) {

x[0] -= DOT\_SIZE;

}

if (rightDirection) {

x[0] += DOT\_SIZE;

}

if (upDirection) {

y[0] -= DOT\_SIZE;

}

if (downDirection) {

y[0] += DOT\_SIZE;

}

}

private void checkCollision() {

for (int z = dots; z > 0; z--) {

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

inGame = false;

}

}

if (y[0] >= B\_HEIGHT) {

inGame = false;

}

if (y[0] < 0) {

inGame = false;

}

if (x[0] >= B\_WIDTH) {

inGame = false;

}

if (x[0] < 0) {

inGame = false;

}

if(!inGame) {

timer.stop();

}

}

private void locateApple() {

int r = (int) (Math.random() \* RAND\_POS);

apple\_x = ((r \* DOT\_SIZE));

r = (int) (Math.random() \* RAND\_POS);

apple\_y = ((r \* DOT\_SIZE));

}

@Override

public void actionPerformed(ActionEvent e) {

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) && (!rightDirection)) {

leftDirection = true;

upDirection = false;

downDirection = false;

}

if ((key == KeyEvent.VK\_RIGHT) && (!leftDirection)) {

rightDirection = true;

upDirection = false;

downDirection = false;

}

if ((key == KeyEvent.VK\_UP) && (!downDirection)) {

upDirection = true;

rightDirection = false;

leftDirection = false;

}

if ((key == KeyEvent.VK\_DOWN) && (!upDirection)) {

downDirection = true;

rightDirection = false;

leftDirection = false;

}

}

}

}

First we will define the constants used in our game.

private final int B\_WIDTH = 300;

private final int B\_HEIGHT = 300;

private final int DOT\_SIZE = 10;

private final int ALL\_DOTS = 900;

private final int RAND\_POS = 29;

private final int DELAY = 140;

The B\_WIDTH and B\_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. The DELAY constant determines the speed of the game.

private final int x[] = new int[ALL\_DOTS];

private final int y[] = new int[ALL\_DOTS];

These two arrays store the x and y coordinates of all joints of a snake.

private void loadImages() {

ImageIcon iid = new ImageIcon("dot.png");

ball = iid.getImage();

ImageIcon iia = new ImageIcon("apple.png");

apple = iia.getImage();

ImageIcon iih = new ImageIcon("head.png");

head = iih.getImage();

}

In the loadImages() method we get the images for the game. The ImageIcon class is used for displaying PNG images.

private void initGame() {

dots = 3;

for (int z = 0; z < dots; z++) {

x[z] = 50 - z \* 10;

y[z] = 50;

}

locateApple();

timer = new Timer(DELAY, this);

timer.start();

}

In the initGame() method we create the snake, randomly locate an apple on the board, and start the timer.

private void checkApple() {

if ((x[0] == apple\_x) && (y[0] == apple\_y)) {

dots++;

locateApple();

}

}

If the apple collides with the head, we increase the number of joints of the snake. We call the locateApple() method which randomly positions a new apple object.

In the move() method we have the key algorithm of the game. To understand it, look at how the snake is moving. We control the head of the snake. We can change its direction with the cursor keys. The rest of the joints move one position up the chain. The second joint moves where the first was, the third joint where the second was etc.

for (int z = dots; z > 0; z--) {

x[z] = x[(z - 1)];

y[z] = y[(z - 1)];

}

This code moves the joints up the chain.

if (leftDirection) {

x[0] -= DOT\_SIZE;

}

This line moves the head to the left.

In the checkCollision() method, we determine if the snake has hit itself or one of the walls.

for (int z = dots; z > 0; z--) {

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

inGame = false;

}

}

If the snake hits one of its joints with its head the game is over.

if (y[0] >= B\_HEIGHT) {

inGame = false;

}

The game is finished if the snake hits the bottom of the board.

Snake.java

package com.zetcode;

import java.awt.EventQueue;

import javax.swing.JFrame;

public class Snake extends JFrame {

public Snake() {

add(new Board());

setResizable(false);

pack();

setTitle("Snake");

setLocationRelativeTo(null);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

public static void main(String[] args) {

EventQueue.invokeLater(new Runnable() {

@Override

public void run() {

JFrame ex = new Snake();

ex.setVisible(true);

}

});

}

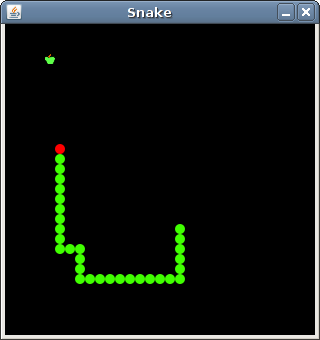
}

This is the main class.

setResizable(false);

pack();

The setResizable() method affects the insets of the JFrame container on some platforms. Therefore, it is important to call it before the pack() method. Otherwise, the collision of the snake's head with the right and bottom borders might not work correctly.

Figure: Snake

This was the Snake game in Java.

**The Breakout game**

In this part of the Java 2D games tutorial we will create a simple Breakout game clone.

The Breakout is an arcade game originally developed by Atari Inc. The game was created in 1976. In this game, the player moves a paddle on the screen and bounces a ball or balls. The objective is to destroy bricks in the top of the window.

**Development**

In our game, we have one paddle, one ball and 30 bricks. I have created an image for a ball, paddle and a brick in Inkscape. We use a timer to create a game cycle. We do not work with angles, we simply change directions. Top, bottom, left and right. I was inspired by the pybreakout game. It was developed in PyGame library by Nathan Dawson.

The game consists of seven files: Commons.java, Sprite.java, Ball.java, Paddle.java, Brick.java, Board.java, and Breakout.java.

Commons.java

package com.zetcode;

public interface Commons {

public static final int WIDTH = 300;

public static final int HEIGTH = 400;

public static final int BOTTOM\_EDGE = 390;

public static final int N\_OF\_BRICKS = 30;

public static final int INIT\_PADDLE\_X = 200;

public static final int INIT\_PADDLE\_Y = 360;

public static final int INIT\_BALL\_X = 230;

public static final int INIT\_BALL\_Y = 355;

public static final int DELAY = 1000;

public static final int PERIOD = 10;

}

The Commons.java file has some common constants. The WIDTH and HEIGHT constants store the dimensions of the board. When the ball passes the BOTTOM\_EDGE, the game is over. The N\_OF\_BRICKS is the number of bricks in the game. The INIT\_PADDLE\_X and INIT\_PADDLE\_Y are initial coordinates of the paddle object. The INIT\_BALL\_X and INIT\_BALL\_Y are initial coordinates of the ball object. The DELAY is the initial delay in milliseconds before task is to be executed and the PERIOD is the time in milliseconds between successive task executions that form game cycles.

Sprite.java

package com.zetcode;

import java.awt.Image;

import java.awt.Rectangle;

public class Sprite {

protected int x;

protected int y;

protected int i\_width;

protected int i\_heigth;

protected Image image;

public void setX(int x) {

this.x = x;

}

public int getX() {

return x;

}

public void setY(int y) {

this.y = y;

}

public int getY() {

return y;

}

public int getWidth() {

return i\_width;

}

public int getHeight() {

return i\_heigth;

}

Image getImage() {

return image;

}

Rectangle getRect() {

return new Rectangle(x, y,

image.getWidth(null), image.getHeight(null));

}

}

The Sprite class is a base class for all objects in the Board. We put here all methods and variables that are in Ball, Brick, and Paddle objects, like getImage() or getX() methods.

Brick.java

package com.zetcode;

import javax.swing.ImageIcon;

public class Brick extends Sprite {

private boolean destroyed;

public Brick(int x, int y) {

this.x = x;

this.y = y;

ImageIcon ii = new ImageIcon("brick.png");

image = ii.getImage();

i\_width = image.getWidth(null);

i\_heigth = image.getHeight(null);

destroyed = false;

}

public boolean isDestroyed() {

return destroyed;

}

public void setDestroyed(boolean val) {

destroyed = val;

}

}

This is the Brick class.

private boolean destroyed;

In the destroyed variable we keep the state of a brick.

Ball.java

package com.zetcode;

import javax.swing.ImageIcon;

public class Ball extends Sprite implements Commons {

private int xdir;

private int ydir;

public Ball() {

xdir = 1;

ydir = -1;

ImageIcon ii = new ImageIcon("ball.png");

image = ii.getImage();

i\_width = image.getWidth(null);

i\_heigth = image.getHeight(null);

resetState();

}

public void move() {

x += xdir;

y += ydir;

if (x == 0) {

setXDir(1);

}

if (x == WIDTH - i\_width) {

setXDir(-1);

}

if (y == 0) {

setYDir(1);

}

}

private void resetState() {

x = INIT\_BALL\_X;

y = INIT\_BALL\_Y;

}

public void setXDir(int x) {

xdir = x;

}

public void setYDir(int y) {

ydir = y;

}

public int getYDir() {

return ydir;

}

}

This is the Ball class.

public void move() {

x += xdir;

y += ydir;

if (x == 0) {

setXDir(1);

}

if (x == WIDTH - i\_width) {

setXDir(-1);

}

if (y == 0) {

setYDir(1);

}

}

The move() method moves the ball on the Board. If the ball hits the borders, the directions are changed accordingly.

public void setXDir(int x) {

xdir = x;

}

public void setYDir(int y) {

ydir = y;

}

These two methods are called when the ball hits the paddle or a brick.

Paddle.java

package com.zetcode;

import java.awt.event.KeyEvent;

import javax.swing.ImageIcon;

public class Paddle extends Sprite implements Commons {

private int dx;

public Paddle() {

ImageIcon ii = new ImageIcon("paddle.png");

image = ii.getImage();

i\_width = image.getWidth(null);

i\_heigth = image.getHeight(null);

resetState();

}

public void move() {

x += dx;

if (x <= 0) {

x = 0;

}

if (x >= WIDTH - i\_width) {

x = WIDTH - i\_width;

}

}

public void keyPressed(KeyEvent e) {

int key = e.getKeyCode();

if (key == KeyEvent.VK\_LEFT) {

dx = -1;

}

if (key == KeyEvent.VK\_RIGHT) {

dx = 1;

}

}

public void keyReleased(KeyEvent e) {

int key = e.getKeyCode();

if (key == KeyEvent.VK\_LEFT) {

dx = 0;

}

if (key == KeyEvent.VK\_RIGHT) {

dx = 0;

}

}

private void resetState() {

x = INIT\_PADDLE\_X;

y = INIT\_PADDLE\_Y;

}

}

This is the Paddle class. It encapsulates the paddle object in the Breakout game. The paddle is controlled with left and right arrow keys. By pressing the arrow key, we set the direction variable. By releasing the arrow key, we set the dx variable to zero. This way the paddle stops moving.

public void move() {

x += dx;

if (x <= 0) {

x = 0;

}

if (x >= WIDTH - i\_width) {

x = WIDTH - i\_width;

}

}

The paddle moves only in the horizontal direction, so we only update the x coordinate. The if conditions ensure that the paddle does not pass the window edges.

Board.java

package com.zetcode;

import java.awt.Color;

import java.awt.Font;

import java.awt.FontMetrics;

import java.awt.Graphics;

import java.awt.Graphics2D;

import java.awt.Point;

import java.awt.RenderingHints;

import java.awt.Toolkit;

import java.awt.event.KeyAdapter;

import java.awt.event.KeyEvent;

import java.util.Timer;

import java.util.TimerTask;

import javax.swing.JPanel;

public class Board extends JPanel implements Commons {

private Timer timer;

private String message = "Game Over";

private Ball ball;

private Paddle paddle;

private Brick bricks[];

private boolean ingame = true;

public Board() {

initBoard();

}

private void initBoard() {

addKeyListener(new TAdapter());

setFocusable(true);

bricks = new Brick[N\_OF\_BRICKS];

setDoubleBuffered(true);

timer = new Timer();

timer.scheduleAtFixedRate(new ScheduleTask(), DELAY, PERIOD);

}

@Override

public void addNotify() {

super.addNotify();

gameInit();

}

private void gameInit() {

ball = new Ball();

paddle = new Paddle();

int k = 0;

for (int i = 0; i < 5; i++) {

for (int j = 0; j < 6; j++) {

bricks[k] = new Brick(j \* 40 + 30, i \* 10 + 50);

k++;

}

}

}

@Override

public void paintComponent(Graphics g) {

super.paintComponent(g);

Graphics2D g2d = (Graphics2D) g;

g2d.setRenderingHint(RenderingHints.KEY\_ANTIALIASING,

RenderingHints.VALUE\_ANTIALIAS\_ON);

g2d.setRenderingHint(RenderingHints.KEY\_RENDERING,

RenderingHints.VALUE\_RENDER\_QUALITY);

if (ingame) {

drawObjects(g2d);

} else {

gameFinished(g2d);

}

Toolkit.getDefaultToolkit().sync();

}

private void drawObjects(Graphics2D g2d) {

g2d.drawImage(ball.getImage(), ball.getX(), ball.getY(),

ball.getWidth(), ball.getHeight(), this);

g2d.drawImage(paddle.getImage(), paddle.getX(), paddle.getY(),

paddle.getWidth(), paddle.getHeight(), this);

for (int i = 0; i < N\_OF\_BRICKS; i++) {

if (!bricks[i].isDestroyed()) {

g2d.drawImage(bricks[i].getImage(), bricks[i].getX(),

bricks[i].getY(), bricks[i].getWidth(),

bricks[i].getHeight(), this);

}

}

}

private void gameFinished(Graphics2D g2d) {

Font font = new Font("Verdana", Font.BOLD, 18);

FontMetrics metr = this.getFontMetrics(font);

g2d.setColor(Color.BLACK);

g2d.setFont(font);

g2d.drawString(message,

(Commons.WIDTH - metr.stringWidth(message)) / 2,

Commons.WIDTH / 2);

}

private class TAdapter extends KeyAdapter {

@Override

public void keyReleased(KeyEvent e) {

paddle.keyReleased(e);

}

@Override

public void keyPressed(KeyEvent e) {

paddle.keyPressed(e);

}

}

private class ScheduleTask extends TimerTask {

@Override

public void run() {

ball.move();

paddle.move();

checkCollision();

repaint();

}

}

private void stopGame() {

ingame = false;

timer.cancel();

}

private void checkCollision() {

if (ball.getRect().getMaxY() > Commons.BOTTOM\_EDGE) {

stopGame();

}

for (int i = 0, j = 0; i < N\_OF\_BRICKS; i++) {

if (bricks[i].isDestroyed()) {

j++;

}

if (j == N\_OF\_BRICKS) {

message = "Victory";

stopGame();

}

}

if ((ball.getRect()).intersects(paddle.getRect())) {

int paddleLPos = (int) paddle.getRect().getMinX();

int ballLPos = (int) ball.getRect().getMinX();

int first = paddleLPos + 8;

int second = paddleLPos + 16;

int third = paddleLPos + 24;

int fourth = paddleLPos + 32;

if (ballLPos < first) {

ball.setXDir(-1);

ball.setYDir(-1);

}

if (ballLPos >= first && ballLPos < second) {

ball.setXDir(-1);

ball.setYDir(-1 \* ball.getYDir());

}

if (ballLPos >= second && ballLPos < third) {

ball.setXDir(0);

ball.setYDir(-1);

}

if (ballLPos >= third && ballLPos < fourth) {

ball.setXDir(1);

ball.setYDir(-1 \* ball.getYDir());

}

if (ballLPos > fourth) {

ball.setXDir(1);

ball.setYDir(-1);

}

}

for (int i = 0; i < N\_OF\_BRICKS; i++) {

if ((ball.getRect()).intersects(bricks[i].getRect())) {

int ballLeft = (int) ball.getRect().getMinX();

int ballHeight = (int) ball.getRect().getHeight();

int ballWidth = (int) ball.getRect().getWidth();

int ballTop = (int) ball.getRect().getMinY();

Point pointRight = new Point(ballLeft + ballWidth + 1, ballTop);

Point pointLeft = new Point(ballLeft - 1, ballTop);

Point pointTop = new Point(ballLeft, ballTop - 1);

Point pointBottom = new Point(ballLeft, ballTop + ballHeight + 1);

if (!bricks[i].isDestroyed()) {

if (bricks[i].getRect().contains(pointRight)) {

ball.setXDir(-1);

} else if (bricks[i].getRect().contains(pointLeft)) {

ball.setXDir(1);

}

if (bricks[i].getRect().contains(pointTop)) {

ball.setYDir(1);

} else if (bricks[i].getRect().contains(pointBottom)) {

ball.setYDir(-1);

}

bricks[i].setDestroyed(true);

}

}

}

}

}

This is the Board class. Here we put the game logic.

public void gameInit() {

ball = new Ball();

paddle = new Paddle();

int k = 0;

for (int i = 0; i < 5; i++) {

for (int j = 0; j < 6; j++) {

bricks[k] = new Brick(j \* 40 + 30, i \* 10 + 50);

k++;

}

}

}

In the gameInit() method we create a ball, a paddle, and thirty bricks.

if (ingame) {

drawObjects(g2d);

} else {

gameFinished(g2d);

}

Depending on the ingame variable, we either draw all the objects in the drawObjects() method or finish the game with the gameFinished() method.

private void drawObjects(Graphics2D g2d) {

g2d.drawImage(ball.getImage(), ball.getX(), ball.getY(),

ball.getWidth(), ball.getHeight(), this);

g2d.drawImage(paddle.getImage(), paddle.getX(), paddle.getY(),

paddle.getWidth(), paddle.getHeight(), this);

for (int i = 0; i < N\_OF\_BRICKS; i++) {

if (!bricks[i].isDestroyed()) {

g2d.drawImage(bricks[i].getImage(), bricks[i].getX(),

bricks[i].getY(), bricks[i].getWidth(),

bricks[i].getHeight(), this);

}

}

}

The drawObjects() method draws all the objects of the game. The sprites are drawn with the drawImage() method.

private void gameFinished(Graphics2D g2d) {

Font font = new Font("Verdana", Font.BOLD, 18);

FontMetrics metr = this.getFontMetrics(font);

g2d.setColor(Color.BLACK);

g2d.setFont(font);

g2d.drawString(message,

(Commons.WIDTH - metr.stringWidth(message)) / 2,

Commons.WIDTH / 2);

}

The gameFinished() method draws "Game over" or "Victory" to the middle of the window.

private class ScheduleTask extends TimerTask {

@Override

public void run() {

ball.move();

paddle.move();

checkCollision();

repaint();

}

}

The ScheduleTask is triggerd every PERIOD ms. In its run() method, we move the ball and the paddle. We check for possible collisions and repaint the screen.

if (ball.getRect().getMaxY() > Commons.BOTTOM\_EDGE) {

stopGame();

}

If the ball hits the bottom, we stop the game.

for (int i = 0, j = 0; i < N\_OF\_BRICKS; i++) {

if (bricks[i].isDestroyed()) {

j++;

}

if (j == N\_OF\_BRICKS) {

message = "Victory";

stopGame();

}

}

We check how many bricks are destroyed. If we destroyed all N\_OF\_BRICKS bricks, we win the game.

if (ballLPos < first) {

ball.setXDir(-1);

ball.setYDir(-1);

}

If the ball hits the first part of the paddle, we change the direction of the ball to the north-west.

if (bricks[i].getRect().contains(pointTop)) {

ball.setYDir(1);

}

If the ball hits the bottom of the brick, we change the y direction of the ball; it goes down.

Breakout.java

package com.zetcode;

import java.awt.EventQueue;

import javax.swing.JFrame;

public class Breakout extends JFrame {

public Breakout() {

initUI();

}

private void initUI() {

add(new Board());

setTitle("Breakout");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

setSize(Commons.WIDTH, Commons.HEIGTH);

setLocationRelativeTo(null);

setResizable(false);

setVisible(true);

}

public static void main(String[] args) {

EventQueue.invokeLater(new Runnable() {

@Override

public void run() {

Breakout game = new Breakout();

game.setVisible(true);

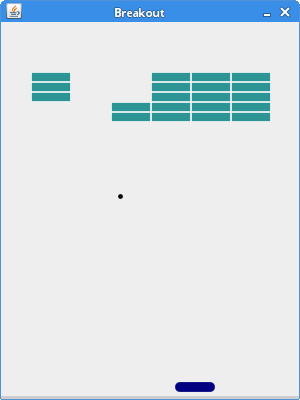
}

});

}

}

And finally, this is the Breakout class which has the main entry method.

Figure: The Breakout game

This was the Breakout game.

# Tetris

In this chapter, we will create a Tetris game clone in Java Swing. It is modified and simplified.

## Tetris

The Tetris game 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. Even my mobile phone has a modified version of the Tetris game.

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: Tetrominoes

## The development

We do not have images for our tetris game, we draw the tetrominoes using Swing drawing API. Behind every computer game, there is a mathematical model. So it is in Tetris.

Some ideas behind the game.

* We use a Timer class to create a game cycle
* The tetrominoes are drawn
* The shapes move on a square by square basis (not pixel by pixel)
* Mathematically a board is a simple list of numbers

I have simplified the game a bit, so that it is easier to understand. The game starts immediately, after it is launched. We can pause the game by pressing the p key. The space key will drop the tetris piece immediately to the bottom. The d key will drop 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.

Tetris.java

package tetris;

import java.awt.BorderLayout;

import javax.swing.JFrame;

import javax.swing.JLabel;

public class Tetris extends JFrame {

JLabel statusbar;

public Tetris() {

statusbar = new JLabel(" 0");

add(statusbar, BorderLayout.SOUTH);

Board board = new Board(this);

add(board);

board.start();

setSize(200, 400);

setTitle("Tetris");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

}

public JLabel getStatusBar() {

return statusbar;

}

public static void main(String[] args) {

Tetris game = new Tetris();

game.setLocationRelativeTo(null);

game.setVisible(true);

}

}

In the Tetris.java file, we set up the game. We create a board on which we play the game. We create a statusbar.

board.start();

The start() method starts the Tetris game. Immediately, after the window appears on the screen.

Shape.java

package tetris;

import java.util.Random;

import java.lang.Math;

public class Shape {

enum Tetrominoes { NoShape, ZShape, SShape, LineShape,

TShape, SquareShape, LShape, MirroredLShape };

private Tetrominoes pieceShape;

private int coords[][];

private int[][][] coordsTable;

public Shape() {

coords = new int[4][2];

setShape(Tetrominoes.NoShape);

}

public void setShape(Tetrominoes shape) {

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 } }

};

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 Tetrominoes getShape() { return pieceShape; }

public void setRandomShape()

{

Random r = new Random();

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

Tetrominoes[] values = Tetrominoes.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 == Tetrominoes.SquareShape)

return this;

Shape 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 == Tetrominoes.SquareShape)

return this;

Shape 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.

enum Tetrominoes { NoShape, ZShape, SShape, LineShape,

TShape, SquareShape, LShape, MirroredLShape };

The Tetrominoes enum holds all seven tetris shapes. Plus the empty shape called here NoShape.

public Shape() {

coords = new int[4][2];

setShape(Tetrominoes.NoShape);

}

This is the constructor of the Shape class. The coords array holds the actual coordinates of a Tetris piece.

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 } }

};

The coordsTable array holds all possible coordinate values of our tetris pieces. This is a template from which all pieces take their coordiate values.

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

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

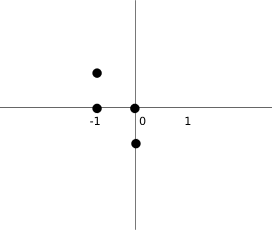
coords[i][j] = coordsTable[shape.ordinal()][i][j];

}

}

Here we put one row of the coordiate values from the coordsTable to a coords array of a tetris piece. Note the use of the ordinal() method. In C++, an enum type is esencially 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.

The following image will help understand the coordinate values a bit more. The coords array saves the coordinates of the tetris piece. For example, numbers { 0, -1 }, { 0, 0 }, { -1, 0 }, { -1, 1 }, represent a rotated S-shape. The following diagram illustrates the shape.

Figure: Coordinates

public Shape rotateLeft()

{

if (pieceShape == Tetrominoes.SquareShape)

return this;

Shape 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 the 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.

Board.java

package tetris;

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;

import javax.swing.JLabel;

import javax.swing.JPanel;

import javax.swing.Timer;

import tetris.Shape.Tetrominoes;

public class Board extends JPanel implements ActionListener {

final int BoardWidth = 10;

final int BoardHeight = 22;

Timer timer;

boolean isFallingFinished = false;

boolean isStarted = false;

boolean isPaused = false;

int numLinesRemoved = 0;

int curX = 0;

int curY = 0;

JLabel statusbar;

Shape curPiece;

Tetrominoes[] board;

public Board(Tetris parent) {

setFocusable(true);

curPiece = new Shape();

timer = new Timer(400, this);

timer.start();

statusbar = parent.getStatusBar();

board = new Tetrominoes[BoardWidth \* BoardHeight];

addKeyListener(new TAdapter());

clearBoard();

}

public void actionPerformed(ActionEvent e) {

if (isFallingFinished) {

isFallingFinished = false;

newPiece();

} else {

oneLineDown();

}

}

int squareWidth() { return (int) getSize().getWidth() / BoardWidth; }

int squareHeight() { return (int) getSize().getHeight() / BoardHeight; }

Tetrominoes shapeAt(int x, int y) { return board[(y \* BoardWidth) + x]; }

public void start()

{

if (isPaused)

return;

isStarted = true;

isFallingFinished = false;

numLinesRemoved = 0;

clearBoard();

newPiece();

timer.start();

}

private void pause()

{

if (!isStarted)

return;

isPaused = !isPaused;

if (isPaused) {

timer.stop();

statusbar.setText("paused");

} else {

timer.start();

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

}

repaint();

}

public void paint(Graphics g)

{

super.paint(g);

Dimension size = getSize();

int boardTop = (int) size.getHeight() - BoardHeight \* squareHeight();

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

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

Tetrominoes shape = shapeAt(j, BoardHeight - i - 1);

if (shape != Tetrominoes.NoShape)

drawSquare(g, 0 + j \* squareWidth(),

boardTop + i \* squareHeight(), shape);

}

}

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

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

int x = curX + curPiece.x(i);

int y = curY - curPiece.y(i);

drawSquare(g, 0 + x \* squareWidth(),

boardTop + (BoardHeight - 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 < BoardHeight \* BoardWidth; ++i)

board[i] = Tetrominoes.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 \* BoardWidth) + x] = curPiece.getShape();

}

removeFullLines();

if (!isFallingFinished)

newPiece();

}

private void newPiece()

{

curPiece.setRandomShape();

curX = BoardWidth / 2 + 1;

curY = BoardHeight - 1 + curPiece.minY();

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

curPiece.setShape(Tetrominoes.NoShape);

timer.stop();

isStarted = false;

statusbar.setText("game over");

}

}

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 >= BoardWidth || y < 0 || y >= BoardHeight)

return false;

if (shapeAt(x, y) != Tetrominoes.NoShape)

return false;

}

curPiece = newPiece;

curX = newX;

curY = newY;

repaint();

return true;

}

private void removeFullLines()

{

int numFullLines = 0;

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

boolean lineIsFull = true;

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

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

lineIsFull = false;

break;

}

}

if (lineIsFull) {

++numFullLines;

for (int k = i; k < BoardHeight - 1; ++k) {

for (int j = 0; j < BoardWidth; ++j)

board[(k \* BoardWidth) + j] = shapeAt(j, k + 1);

}

}

}

if (numFullLines > 0) {

numLinesRemoved += numFullLines;

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

isFallingFinished = true;

curPiece.setShape(Tetrominoes.NoShape);

repaint();

}

}

private void drawSquare(Graphics g, int x, int y, Tetrominoes 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)

};

Color 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);

}

class TAdapter extends KeyAdapter {

public void keyPressed(KeyEvent e) {

if (!isStarted || curPiece.getShape() == Tetrominoes.NoShape) {

return;

}

int keycode = e.getKeyCode();

if (keycode == 'p' || keycode == 'P') {

pause();

return;

}

if (isPaused)

return;

switch (keycode) {

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 'd':

oneLineDown();

break;

case 'D':

oneLineDown();

break;

}

}

}

}

Finally, we have the Board.java file. This is where the game logic is located.

...

isFallingFinished = false;

isStarted = false;

isPaused = false;

numLinesRemoved = 0;

curX = 0;

curY = 0;

...

We initialize some important variables. The isFallingFinished variable determines if the tetris shape has finished falling and we then need to create a new shape. The numLinesRemoved counts the number of lines we have removed so far. The curX and curY variables determine the actual position of the falling tetris shape.

setFocusable(true);

We must explicitly call the setFocusable() method. From now, the board has the keyboard input.

timer = new Timer(400, this);

timer.start();

Timer object fires one or more action events after a specified delay. In our case, the timer calls the actionPerformed() method each 400 ms.

public void actionPerformed(ActionEvent e) {

if (isFallingFinished) {

isFallingFinished = false;

newPiece();

} else {

oneLineDown();

}

}

The actionPerformed() method checks if the falling has finished. If so, a new piece is created. If not, the falling tetris piece goes one line down.

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

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

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

Tetrominoes shape = shapeAt(j, BoardHeight - i - 1);

if (shape != Tetrominoes.NoShape)

drawSquare(g, 0 + 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 rememberd in the board array. We access it using the shapeAt() method.

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

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

int x = curX + curPiece.x(i);

int y = curY - curPiece.y(i);

drawSquare(g, 0 + x \* squareWidth(),

boardTop + (BoardHeight - 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.

private void clearBoard()

{

for (int i = 0; i < BoardHeight \* BoardWidth; ++i)

board[i] = Tetrominoes.NoShape;

}

The clearBoard() method fills the board with empty NoSpapes. 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 \* BoardWidth) + 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. More precisely, we try to create a new piece.

private void newPiece()

{

curPiece.setRandomShape();

curX = BoardWidth / 2 + 1;

curY = BoardHeight - 1 + curPiece.minY();

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

curPiece.setShape(Tetrominoes.NoShape);

timer.stop();

isStarted = false;

statusbar.setText("game over");

}

}

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. We put 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 >= BoardWidth || y < 0 || y >= BoardHeight)

return false;

if (shapeAt(x, y) != Tetrominoes.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.

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

boolean lineIsFull = true;

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

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

lineIsFull = false;

break;

}

}

if (lineIsFull) {

++numFullLines;

for (int k = i; k < BoardHeight - 1; ++k) {

for (int j = 0; j < BoardWidth; ++j)

board[(k \* BoardWidth) + j] = shapeAt(j, k + 1);

}

}

}

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.

Every tetris piece has four squares. Each of the squares is drawn with the drawSquare() method. Tetris pieces have different colours.

g.setColor(color.brighter());

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

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

The left and top sides of a square are drawn with a brighter colour. Similarly, the bottom and right sides are drawn with darker colours. This is to simulate a 3D edge.

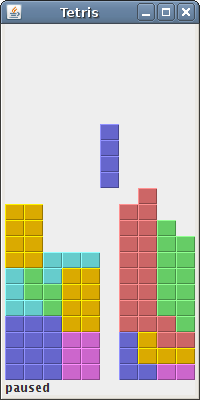
We control the game with a keyboard. The control mechanism is implemented with a KeyAdapter. This is an inner class that overrides the keyPressed() method.

case KeyEvent.VK\_RIGHT:

tryMove(curPiece, curX + 1, curY);

break;

If we pressed the left arrow key, we try to move the falling piece one square to the left.

Figure: Tetris

This was the Tetris game.

# Pacman

In this part of the Java 2D games tutorial we will create a simple Pacman game clone.

Pacman is an arcade game originally developed by a Japanese company Namco in 1980. Pacman became one of the most popular arcade games ever created.

## Development

The following code example is a remake of a Pacman game by Brian Postma available at [http://www.brianpostma.com](http://www.brianpostma.com/java.html). The code is modified and simplified so that it is easier to understand.

The goal of the game is to collect all the points in the maze and avoid the ghosts. The Pacman is animated in two ways: his position in the maze and his body. We animate his body with four images, depending on the direction. The animation is used to create the illusion of Pacman opening and closing his mouth. The maze consists of 15x15 squares. The structure of the maze is based on a simple array of integers. Pacman has three lives. We also count the score.

The game consists of two files: Board.java and Pacman.java.

Board.java

package com.zetcode;

import java.awt.BasicStroke;

import java.awt.Color;

import java.awt.Dimension;

import java.awt.Event;

import java.awt.Font;

import java.awt.FontMetrics;

import java.awt.Graphics;

import java.awt.Graphics2D;

import java.awt.Image;

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 javax.swing.ImageIcon;

import javax.swing.JPanel;

import javax.swing.Timer;

public class Board extends JPanel implements ActionListener {

private Dimension d;

private final Font smallfont = new Font("Helvetica", Font.BOLD, 14);

private Image ii;

private final Color dotcolor = new Color(192, 192, 0);

private Color mazecolor;

private boolean ingame = false;

private boolean dying = false;

private final int blocksize = 24;

private final int nrofblocks = 15;

private final int scrsize = nrofblocks \* blocksize;

private final int pacanimdelay = 2;

private final int pacmananimcount = 4;

private final int maxghosts = 12;

private final int pacmanspeed = 6;

private int pacanimcount = pacanimdelay;

private int pacanimdir = 1;

private int pacmananimpos = 0;

private int nrofghosts = 6;

private int pacsleft, score;

private int[] dx, dy;

private int[] ghostx, ghosty, ghostdx, ghostdy, ghostspeed;

private Image ghost;

private Image pacman1, pacman2up, pacman2left, pacman2right, pacman2down;

private Image pacman3up, pacman3down, pacman3left, pacman3right;

private Image pacman4up, pacman4down, pacman4left, pacman4right;

private int pacmanx, pacmany, pacmandx, pacmandy;

private int reqdx, reqdy, viewdx, viewdy;

private final short leveldata[] = {

19, 26, 26, 26, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 22,

21, 0, 0, 0, 17, 16, 16, 16, 16, 16, 16, 16, 16, 16, 20,

21, 0, 0, 0, 17, 16, 16, 16, 16, 16, 16, 16, 16, 16, 20,

21, 0, 0, 0, 17, 16, 16, 24, 16, 16, 16, 16, 16, 16, 20,

17, 18, 18, 18, 16, 16, 20, 0, 17, 16, 16, 16, 16, 16, 20,

17, 16, 16, 16, 16, 16, 20, 0, 17, 16, 16, 16, 16, 24, 20,

25, 16, 16, 16, 24, 24, 28, 0, 25, 24, 24, 16, 20, 0, 21,

1, 17, 16, 20, 0, 0, 0, 0, 0, 0, 0, 17, 20, 0, 21,

1, 17, 16, 16, 18, 18, 22, 0, 19, 18, 18, 16, 20, 0, 21,

1, 17, 16, 16, 16, 16, 20, 0, 17, 16, 16, 16, 20, 0, 21,

1, 17, 16, 16, 16, 16, 20, 0, 17, 16, 16, 16, 20, 0, 21,

1, 17, 16, 16, 16, 16, 16, 18, 16, 16, 16, 16, 20, 0, 21,

1, 17, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 20, 0, 21,

1, 25, 24, 24, 24, 24, 24, 24, 24, 24, 16, 16, 16, 18, 20,

9, 8, 8, 8, 8, 8, 8, 8, 8, 8, 25, 24, 24, 24, 28

};

private final int validspeeds[] = {1, 2, 3, 4, 6, 8};

private final int maxspeed = 6;

private int currentspeed = 3;

private short[] screendata;

private Timer timer;

public Board() {

loadImages();

initVariables();

addKeyListener(new TAdapter());

setFocusable(true);

setBackground(Color.black);

setDoubleBuffered(true);

}

private void initVariables() {

screendata = new short[nrofblocks \* nrofblocks];

mazecolor = new Color(5, 100, 5);

d = new Dimension(400, 400);

ghostx = new int[maxghosts];

ghostdx = new int[maxghosts];

ghosty = new int[maxghosts];

ghostdy = new int[maxghosts];

ghostspeed = new int[maxghosts];

dx = new int[4];

dy = new int[4];

timer = new Timer(40, this);

timer.start();

}

@Override

public void addNotify() {

super.addNotify();

initGame();

}

private void doAnim() {

pacanimcount--;

if (pacanimcount <= 0) {

pacanimcount = pacanimdelay;

pacmananimpos = pacmananimpos + pacanimdir;

if (pacmananimpos == (pacmananimcount - 1) || pacmananimpos == 0) {

pacanimdir = -pacanimdir;

}

}

}

private void playGame(Graphics2D g2d) {

if (dying) {

death();

} else {

movePacman();

drawPacman(g2d);

moveGhosts(g2d);

checkMaze();

}

}

private void showIntroScreen(Graphics2D g2d) {

g2d.setColor(new Color(0, 32, 48));

g2d.fillRect(50, scrsize / 2 - 30, scrsize - 100, 50);

g2d.setColor(Color.white);

g2d.drawRect(50, scrsize / 2 - 30, scrsize - 100, 50);

String s = "Press s to start.";

Font small = new Font("Helvetica", Font.BOLD, 14);

FontMetrics metr = this.getFontMetrics(small);

g2d.setColor(Color.white);

g2d.setFont(small);

g2d.drawString(s, (scrsize - metr.stringWidth(s)) / 2, scrsize / 2);

}

private void drawScore(Graphics2D g) {

int i;

String s;

g.setFont(smallfont);

g.setColor(new Color(96, 128, 255));

s = "Score: " + score;

g.drawString(s, scrsize / 2 + 96, scrsize + 16);

for (i = 0; i < pacsleft; i++) {

g.drawImage(pacman3left, i \* 28 + 8, scrsize + 1, this);

}

}

private void checkMaze() {

short i = 0;

boolean finished = true;

while (i < nrofblocks \* nrofblocks && finished) {

if ((screendata[i] & 48) != 0) {

finished = false;

}

i++;

}

if (finished) {

score += 50;

if (nrofghosts < maxghosts) {

nrofghosts++;

}

if (currentspeed < maxspeed) {

currentspeed++;

}

initLevel();

}

}

private void death() {

pacsleft--;

if (pacsleft == 0) {

ingame = false;

}

continueLevel();

}

private void moveGhosts(Graphics2D g2d) {

short i;

int pos;

int count;

for (i = 0; i < nrofghosts; i++) {

if (ghostx[i] % blocksize == 0 && ghosty[i] % blocksize == 0) {

pos = ghostx[i] / blocksize + nrofblocks \* (int) (ghosty[i] / blocksize);

count = 0;

if ((screendata[pos] & 1) == 0 && ghostdx[i] != 1) {

dx[count] = -1;

dy[count] = 0;

count++;

}

if ((screendata[pos] & 2) == 0 && ghostdy[i] != 1) {

dx[count] = 0;

dy[count] = -1;

count++;

}

if ((screendata[pos] & 4) == 0 && ghostdx[i] != -1) {

dx[count] = 1;

dy[count] = 0;

count++;

}

if ((screendata[pos] & 8) == 0 && ghostdy[i] != -1) {

dx[count] = 0;

dy[count] = 1;

count++;

}

if (count == 0) {

if ((screendata[pos] & 15) == 15) {

ghostdx[i] = 0;

ghostdy[i] = 0;

} else {

ghostdx[i] = -ghostdx[i];

ghostdy[i] = -ghostdy[i];

}

} else {

count = (int) (Math.random() \* count);

if (count > 3) {

count = 3;

}

ghostdx[i] = dx[count];

ghostdy[i] = dy[count];

}

}

ghostx[i] = ghostx[i] + (ghostdx[i] \* ghostspeed[i]);

ghosty[i] = ghosty[i] + (ghostdy[i] \* ghostspeed[i]);

drawGhost(g2d, ghostx[i] + 1, ghosty[i] + 1);

if (pacmanx > (ghostx[i] - 12) && pacmanx < (ghostx[i] + 12)

&& pacmany > (ghosty[i] - 12) && pacmany < (ghosty[i] + 12)

&& ingame) {

dying = true;

}

}

}

private void drawGhost(Graphics2D g2d, int x, int y) {

g2d.drawImage(ghost, x, y, this);

}

private void movePacman() {

int pos;

short ch;

if (reqdx == -pacmandx && reqdy == -pacmandy) {

pacmandx = reqdx;

pacmandy = reqdy;

viewdx = pacmandx;

viewdy = pacmandy;

}

if (pacmanx % blocksize == 0 && pacmany % blocksize == 0) {

pos = pacmanx / blocksize + nrofblocks \* (int) (pacmany / blocksize);

ch = screendata[pos];

if ((ch & 16) != 0) {

screendata[pos] = (short) (ch & 15);

score++;

}

if (reqdx != 0 || reqdy != 0) {

if (!((reqdx == -1 && reqdy == 0 && (ch & 1) != 0)

|| (reqdx == 1 && reqdy == 0 && (ch & 4) != 0)

|| (reqdx == 0 && reqdy == -1 && (ch & 2) != 0)

|| (reqdx == 0 && reqdy == 1 && (ch & 8) != 0))) {

pacmandx = reqdx;

pacmandy = reqdy;

viewdx = pacmandx;

viewdy = pacmandy;

}

}

// Check for standstill

if ((pacmandx == -1 && pacmandy == 0 && (ch & 1) != 0)

|| (pacmandx == 1 && pacmandy == 0 && (ch & 4) != 0)

|| (pacmandx == 0 && pacmandy == -1 && (ch & 2) != 0)

|| (pacmandx == 0 && pacmandy == 1 && (ch & 8) != 0)) {

pacmandx = 0;

pacmandy = 0;

}

}

pacmanx = pacmanx + pacmanspeed \* pacmandx;

pacmany = pacmany + pacmanspeed \* pacmandy;

}

private void drawPacman(Graphics2D g2d) {

if (viewdx == -1) {

drawPacnanLeft(g2d);

} else if (viewdx == 1) {

drawPacmanRight(g2d);

} else if (viewdy == -1) {

drawPacmanUp(g2d);

} else {

drawPacmanDown(g2d);

}

}

private void drawPacmanUp(Graphics2D g2d) {

switch (pacmananimpos) {

case 1:

g2d.drawImage(pacman2up, pacmanx + 1, pacmany + 1, this);

break;

case 2:

g2d.drawImage(pacman3up, pacmanx + 1, pacmany + 1, this);

break;

case 3:

g2d.drawImage(pacman4up, pacmanx + 1, pacmany + 1, this);

break;

default:

g2d.drawImage(pacman1, pacmanx + 1, pacmany + 1, this);

break;

}

}

private void drawPacmanDown(Graphics2D g2d) {

switch (pacmananimpos) {

case 1:

g2d.drawImage(pacman2down, pacmanx + 1, pacmany + 1, this);

break;

case 2:

g2d.drawImage(pacman3down, pacmanx + 1, pacmany + 1, this);

break;

case 3:

g2d.drawImage(pacman4down, pacmanx + 1, pacmany + 1, this);

break;

default:

g2d.drawImage(pacman1, pacmanx + 1, pacmany + 1, this);

break;

}

}

private void drawPacnanLeft(Graphics2D g2d) {

switch (pacmananimpos) {

case 1:

g2d.drawImage(pacman2left, pacmanx + 1, pacmany + 1, this);

break;

case 2:

g2d.drawImage(pacman3left, pacmanx + 1, pacmany + 1, this);

break;

case 3:

g2d.drawImage(pacman4left, pacmanx + 1, pacmany + 1, this);

break;

default:

g2d.drawImage(pacman1, pacmanx + 1, pacmany + 1, this);

break;

}

}

private void drawPacmanRight(Graphics2D g2d) {

switch (pacmananimpos) {

case 1:

g2d.drawImage(pacman2right, pacmanx + 1, pacmany + 1, this);

break;

case 2:

g2d.drawImage(pacman3right, pacmanx + 1, pacmany + 1, this);

break;

case 3:

g2d.drawImage(pacman4right, pacmanx + 1, pacmany + 1, this);

break;

default:

g2d.drawImage(pacman1, pacmanx + 1, pacmany + 1, this);

break;

}

}

private void drawMaze(Graphics2D g2d) {

short i = 0;

int x, y;

for (y = 0; y < scrsize; y += blocksize) {

for (x = 0; x < scrsize; x += blocksize) {

g2d.setColor(mazecolor);

g2d.setStroke(new BasicStroke(2));

if ((screendata[i] & 1) != 0) {

g2d.drawLine(x, y, x, y + blocksize - 1);

}

if ((screendata[i] & 2) != 0) {

g2d.drawLine(x, y, x + blocksize - 1, y);

}

if ((screendata[i] & 4) != 0) {

g2d.drawLine(x + blocksize - 1, y, x + blocksize - 1,

y + blocksize - 1);

}

if ((screendata[i] & 8) != 0) {

g2d.drawLine(x, y + blocksize - 1, x + blocksize - 1,

y + blocksize - 1);

}

if ((screendata[i] & 16) != 0) {

g2d.setColor(dotcolor);

g2d.fillRect(x + 11, y + 11, 2, 2);

}

i++;

}

}

}

private void initGame() {

pacsleft = 3;

score = 0;

initLevel();

nrofghosts = 6;

currentspeed = 3;

}

private void initLevel() {

int i;

for (i = 0; i < nrofblocks \* nrofblocks; i++) {

screendata[i] = leveldata[i];

}

continueLevel();

}

private void continueLevel() {

short i;

int dx = 1;

int random;

for (i = 0; i < nrofghosts; i++) {

ghosty[i] = 4 \* blocksize;

ghostx[i] = 4 \* blocksize;

ghostdy[i] = 0;

ghostdx[i] = dx;

dx = -dx;

random = (int) (Math.random() \* (currentspeed + 1));

if (random > currentspeed) {

random = currentspeed;

}

ghostspeed[i] = validspeeds[random];

}

pacmanx = 7 \* blocksize;

pacmany = 11 \* blocksize;

pacmandx = 0;

pacmandy = 0;

reqdx = 0;

reqdy = 0;

viewdx = -1;

viewdy = 0;

dying = false;

}

private void loadImages() {

ghost = new ImageIcon("images/ghost.png").getImage();

pacman1 = new ImageIcon("images/pacman.png").getImage();

pacman2up = new ImageIcon("images/up1.png").getImage();

pacman3up = new ImageIcon("images/up2.png").getImage();

pacman4up = new ImageIcon("images/up3.png").getImage();

pacman2down = new ImageIcon("images/down1.png").getImage();

pacman3down = new ImageIcon("images/down2.png").getImage();

pacman4down = new ImageIcon("images/down3.png").getImage();

pacman2left = new ImageIcon("images/left1.png").getImage();

pacman3left = new ImageIcon("images/left2.png").getImage();

pacman4left = new ImageIcon("images/left3.png").getImage();

pacman2right = new ImageIcon("images/right1.png").getImage();

pacman3right = new ImageIcon("images/right2.png").getImage();

pacman4right = new ImageIcon("images/right3.png").getImage();

}

@Override

public void paintComponent(Graphics g) {

super.paintComponent(g);

doDrawing(g);

}

private void doDrawing(Graphics g) {

Graphics2D g2d = (Graphics2D) g;

g2d.setColor(Color.black);

g2d.fillRect(0, 0, d.width, d.height);

drawMaze(g2d);

drawScore(g2d);

doAnim();

if (ingame) {

playGame(g2d);

} else {

showIntroScreen(g2d);

}

g2d.drawImage(ii, 5, 5, this);

Toolkit.getDefaultToolkit().sync();

g2d.dispose();

}

class TAdapter extends KeyAdapter {

@Override

public void keyPressed(KeyEvent e) {

int key = e.getKeyCode();

if (ingame) {

if (key == KeyEvent.VK\_LEFT) {

reqdx = -1;

reqdy = 0;

} else if (key == KeyEvent.VK\_RIGHT) {

reqdx = 1;

reqdy = 0;

} else if (key == KeyEvent.VK\_UP) {

reqdx = 0;

reqdy = -1;

} else if (key == KeyEvent.VK\_DOWN) {

reqdx = 0;

reqdy = 1;

} else if (key == KeyEvent.VK\_ESCAPE && timer.isRunning()) {

ingame = false;

} else if (key == KeyEvent.VK\_PAUSE) {

if (timer.isRunning()) {

timer.stop();

} else {

timer.start();

}

}

} else {

if (key == 's' || key == 'S') {

ingame = true;

initGame();

}

}

}

@Override

public void keyReleased(KeyEvent e) {

int key = e.getKeyCode();

if (key == Event.LEFT || key == Event.RIGHT

|| key == Event.UP || key == Event.DOWN) {

reqdx = 0;

reqdy = 0;

}

}

}

@Override

public void actionPerformed(ActionEvent e) {

repaint();

}

}

The Pacman is controlled with the cursor keys. The Esc key finishes the game, the Pause key pauses it.

private int pacmanx, pacmany, pacmandx, pacmandy;

The first two variables store the x and y coordinates of the Pacman sprite. The last two variables are the delta changes in horizontal and vertical directions.

private final short leveldata[] = {

19, 26, 26, 26, 18, 18, 18, 18, ...

};

These numbers make up the maze. They provide information out of which we create the corners and the points. Number 1 is a left corner. Numbers 2, 4 and 8 represent top, right, bottom corners respectively. Number 16 is a point. These number can be added, for example number 19 in the upper left corner means that the square will have top and left borders and a point (16 + 2 + 1).

private void doAnim() {

pacanimcount--;

if (pacanimcount <= 0) {

pacanimcount = pacanimdelay;

pacmananimpos = pacmananimpos + pacanimdir;

if (pacmananimpos == (pacmananimcount - 1) || pacmananimpos == 0) {

pacanimdir = -pacanimdir;

}

}

}

The doAnim() counts the pacmananimpos variable which determines what pacman image is drawn. There are four pacman images. There is also a pacanimdelay variable which makes the animation a bit slower. Otherwise the pacman would open his mouth too fast.

boolean finished = true;

while (i < nrofblocks \* nrofblocks && finished) {

if ((screendata[i] & 48) != 0) {

finished = false;

}

i++;

}

This code is part of the checkMaze() method. It checks if there are any points left for the Pacman to eat. Number 16 stands for a point. If all points are consumed, we move to the next level. (In our case, we just restart the game.)

Next we will examine the moveGhosts() method. The ghosts move one square and then decide if they change the direction.

if (ghostx[i] % blocksize == 0 && ghosty[i] % blocksize == 0) {

We continue only if we have finished moving one square.

pos = ghostx[i] / blocksize + nrofblocks \* (int)(ghosty[i] / blocksize);

This line determines where the ghost is situated. In which position/square. There are 225 theoretical positions. (A ghost cannot move over walls.)

if ((screendata[pos] & 1) == 0 && ghostdx[i] != 1) {

dx[count] = -1;

dy[count] = 0;

count++;

}

If there is no obstacle on the left and the ghost is not already moving to the right, the ghost will move to the left. What does this code really mean? If the ghost enters a tunnel, he will continue in the same direction until he is out of the tunnel. Moving of ghosts is partly random. We do not apply this randomness inside long tunnels. The ghost might get stuck there.

if (pacmanx > (ghostx[i] - 12) && pacmanx < (ghostx[i] + 12)

&& pacmany > (ghosty[i] - 12) && pacmany < (ghosty[i] + 12)

&& ingame) {

dying = true;

}

If there is a collision between ghosts and a Pacman, the Pacman dies.

Next we are going to examine the movePacman() method. The reqdx and reqdy variables are determined in the TAdapter inner class. These variables are controlled with cursor keys.

if ((ch & 16) != 0) {

screendata[pos] = (short) (ch & 15);

score++;

}

If the pacman moves to a position with a point, we remove it from the maze and increase the score value.

if ((pacmandx == -1 && pacmandy == 0 && (ch & 1) != 0) ||

(pacmandx == 1 && pacmandy == 0 && (ch & 4) != 0) ||

(pacmandx == 0 && pacmandy == -1 && (ch & 2) != 0) ||

(pacmandx == 0 && pacmandy == 1 && (ch & 8) != 0)) {

pacmandx = 0;

pacmandy = 0;

}

The Pacman stops if he cannot move further it his current direction.

private void drawPacman(Graphics2D g2d) {

if (viewdx == -1) {

drawPacnanLeft(g2d);

} else if (viewdx == 1) {

drawPacmanRight(g2d);

} else if (viewdy == -1) {

drawPacmanUp(g2d);

} else {

drawPacmanDown(g2d);

}

}

There are four possible directions for a Pacman. There are four images for all directions. The images are used to animate Pacman opening and closing his mouth.

The drawMaze() method draws the maze out of the numbers in the screendata array. Number 1 is a left border, 2 is a top border, 4 is a right border, 8 is a bottom border and 16 is a point. We simply go through all 225 squares in the maze. For example we have 9 in the screendata array. We have the first bit (1) and the fourth bit (8) set. So we draw a bottom and a left border on this particular square.

if ((screendata[i] & 1) != 0) {

g2d.drawLine(x, y, x, y + blocksize - 1);

}

We draw a left border if the first bit of a number is set.

Pacman.java

package com.zetcode;

import java.awt.EventQueue;

import javax.swing.JFrame;

public class Pacman extends JFrame {

public Pacman() {

initUI();

}

private void initUI() {

add(new Board());

setTitle("Pacman");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

setSize(380, 420);

setLocationRelativeTo(null);

setVisible(true);

}

public static void main(String[] args) {

EventQueue.invokeLater(new Runnable() {

@Override

public void run() {

Pacman ex = new Pacman();

ex.setVisible(true);

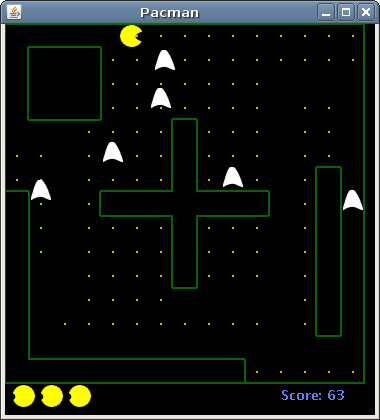
}

});

}

}

This is a Pacman file with a main method.

Figure: Pacman

This was the Pacman game.

# Space Invaders

In this part of the Java 2D games tutorial we will create a simple Space Invaders game clone.

Space Invaders is an arcade video game designed by Tomohiro Nishikado. It was first released in 1978. The player controls a cannon. He is about to save the Earth from invasion of evil space invaders.

## Development

In our Java clone we have 24 invaders. These aliens heavily shell the ground. When the player shoots a missile, he can shoot another one only when it hits an alien or the top of the Board. The player shoots with the Alt key. Aliens launch randomly their bombs. Each alien shoots a bomb only after the previous one hits the bottom.

SpaceInvaders.java

package spaceinvaders;

import javax.swing.JFrame;

public class SpaceInvaders extends JFrame implements Commons {

public SpaceInvaders()

{

add(new Board());

setTitle("Space Invaders");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

setSize(BOARD\_WIDTH, BOARD\_HEIGTH);

setLocationRelativeTo(null);

setVisible(true);

setResizable(false);

}

public static void main(String[] args) {

new SpaceInvaders();

}

}

This is the main class.

Commons.java

package spaceinvaders;

public interface Commons {

public static final int BOARD\_WIDTH = 358;

public static final int BOARD\_HEIGTH = 350;

public static final int GROUND = 290;

public static final int BOMB\_HEIGHT = 5;

public static final int ALIEN\_HEIGHT = 12;

public static final int ALIEN\_WIDTH = 12;

public static final int BORDER\_RIGHT = 30;

public static final int BORDER\_LEFT = 5;

public static final int GO\_DOWN = 15;

public static final int NUMBER\_OF\_ALIENS\_TO\_DESTROY = 24;

public static final int CHANCE = 5;

public static final int DELAY = 17;

public static final int PLAYER\_WIDTH = 15;

public static final int PLAYER\_HEIGHT = 10;

}

The Commons.java file has some common constants. They are self-explanatory.

Alien.java

package spaceinvaders;

import javax.swing.ImageIcon;

public class Alien extends Sprite {

private Bomb bomb;

private final String shot = "../spacepix/alien.png";

public Alien(int x, int y) {

this.x = x;

this.y = y;

bomb = new Bomb(x, y);

ImageIcon ii = new ImageIcon(this.getClass().getResource(shot));

setImage(ii.getImage());

}

public void act(int direction) {

this.x += direction;

}

public Bomb getBomb() {

return bomb;

}

public class Bomb extends Sprite {

private final String bomb = "../spacepix/bomb.png";

private boolean destroyed;

public Bomb(int x, int y) {

setDestroyed(true);

this.x = x;

this.y = y;

ImageIcon ii = new ImageIcon(this.getClass().getResource(bomb));

setImage(ii.getImage());

}

public void setDestroyed(boolean destroyed) {

this.destroyed = destroyed;

}

public boolean isDestroyed() {

return destroyed;

}

}

}

This is the Alien sprite. Each alien has an inner Bomb class.

public void act(int direction) {

this.x += direction;

}

The act() method is called from the Board class. It is used to position an alien in horizontal direction.

public Bomb getBomb() {

return bomb;

}

The getBomb() method is called, when the alien is about to drop a bomb.

Player.java

package spaceinvaders;

import java.awt.event.KeyEvent;

import javax.swing.ImageIcon;

public class Player extends Sprite implements Commons{

private final int START\_Y = 280;

private final int START\_X = 270;

private final String player = "../spacepix/player.png";

private int width;

public Player() {

ImageIcon ii = new ImageIcon(this.getClass().getResource(player));

width = ii.getImage().getWidth(null);

setImage(ii.getImage());

setX(START\_X);

setY(START\_Y);

}

public void act() {

x += dx;

if (x <= 2)

x = 2;

if (x >= BOARD\_WIDTH - 2\*width)

x = BOARD\_WIDTH - 2\*width;

}

public void keyPressed(KeyEvent e) {

int key = e.getKeyCode();

if (key == KeyEvent.VK\_LEFT)

{

dx = -2;

}

if (key == KeyEvent.VK\_RIGHT)

{

dx = 2;

}

}

public void keyReleased(KeyEvent e) {

int key = e.getKeyCode();

if (key == KeyEvent.VK\_LEFT)

{

dx = 0;

}

if (key == KeyEvent.VK\_RIGHT)

{

dx = 0;

}

}

}

This is the Player sprite. We control the cannon with the cursor keys.

private final int START\_Y = 280;

private final int START\_X = 270;

These are the initial coordinates of the player sprite.

if (key == KeyEvent.VK\_LEFT)

{

dx = -2;

}

If we press the left cursor key, the dx variable is set to -2. Next time the act() method is called, the player moves to the left.

if (key == KeyEvent.VK\_LEFT)

{

dx = 0;

}

if (key == KeyEvent.VK\_RIGHT)

{

dx = 0;

}

If we release the left or the right cursor, the dx variable is set to zero. The player sprite stops moving.

Shot.java

package spaceinvaders;

import javax.swing.ImageIcon;

public class Shot extends Sprite {

private String shot = "../spacepix/shot.png";

private final int H\_SPACE = 6;

private final int V\_SPACE = 1;

public Shot() {

}

public Shot(int x, int y) {

ImageIcon ii = new ImageIcon(this.getClass().getResource(shot));

setImage(ii.getImage());

setX(x + H\_SPACE);

setY(y - V\_SPACE);

}

}

This is the Shot sprite. The shot is triggered with the ALT key. The H\_SPACE and the V\_SPACE constants are used to position the missile appropriately.

Sprite.java

package spaceinvaders;

import java.awt.Image;

public class Sprite {

private boolean visible;

private Image image;

protected int x;

protected int y;

protected boolean dying;

protected int dx;

public Sprite() {

visible = true;

}

public void die() {

visible = false;

}

public boolean isVisible() {

return visible;

}

protected void setVisible(boolean visible) {

this.visible = visible;

}

public void setImage(Image image) {

this.image = image;

}

public Image getImage() {

return image;

}

public void setX(int x) {

this.x = x;

}

public void setY(int y) {

this.y = y;

}

public int getY() {

return y;

}

public int getX() {

return x;

}

public void setDying(boolean dying) {

this.dying = dying;

}

public boolean isDying() {

return this.dying;

}

}

This is the basic Sprite class. Other sprites inherit from it. It has some common functionality.

Board.java

package spaceinvaders;

import java.awt.Color;

import java.awt.Dimension;

import java.awt.Font;

import java.awt.FontMetrics;

import java.awt.Graphics;

import java.awt.Toolkit;

import java.awt.event.KeyAdapter;

import java.awt.event.KeyEvent;

import java.util.ArrayList;

import java.util.Iterator;

import java.util.Random;

import javax.swing.ImageIcon;

import javax.swing.JPanel;

public class Board extends JPanel implements Runnable, Commons {

private Dimension d;

private ArrayList aliens;

private Player player;

private Shot shot;

private int alienX = 150;

private int alienY = 5;

private int direction = -1;

private int deaths = 0;

private boolean ingame = true;

private final String expl = "../spacepix/explosion.png";

private final String alienpix = "../spacepix/alien.png";

private String message = "Game Over";

private Thread animator;

public Board()

{

addKeyListener(new TAdapter());

setFocusable(true);

d = new Dimension(BOARD\_WIDTH, BOARD\_HEIGTH);

setBackground(Color.black);

gameInit();

setDoubleBuffered(true);

}

public void addNotify() {

super.addNotify();

gameInit();

}

public void gameInit() {

aliens = new ArrayList();

ImageIcon ii = new ImageIcon(this.getClass().getResource(alienpix));

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

for (int j=0; j < 6; j++) {

Alien alien = new Alien(alienX + 18\*j, alienY + 18\*i);

alien.setImage(ii.getImage());

aliens.add(alien);

}

}

player = new Player();

shot = new Shot();

if (animator == null || !ingame) {

animator = new Thread(this);

animator.start();

}

}

public void drawAliens(Graphics g)

{

Iterator it = aliens.iterator();

while (it.hasNext()) {

Alien alien = (Alien) it.next();

if (alien.isVisible()) {

g.drawImage(alien.getImage(), alien.getX(), alien.getY(), this);

}

if (alien.isDying()) {

alien.die();

}

}

}

public void drawPlayer(Graphics g) {

if (player.isVisible()) {

g.drawImage(player.getImage(), player.getX(), player.getY(), this);

}

if (player.isDying()) {

player.die();

ingame = false;

}

}

public void drawShot(Graphics g) {

if (shot.isVisible())

g.drawImage(shot.getImage(), shot.getX(), shot.getY(), this);

}

public void drawBombing(Graphics g) {

Iterator i3 = aliens.iterator();

while (i3.hasNext()) {

Alien a = (Alien) i3.next();

Alien.Bomb b = a.getBomb();

if (!b.isDestroyed()) {

g.drawImage(b.getImage(), b.getX(), b.getY(), this);

}

}

}

public void paint(Graphics g)

{

super.paint(g);

g.setColor(Color.black);

g.fillRect(0, 0, d.width, d.height);

g.setColor(Color.green);

if (ingame) {

g.drawLine(0, GROUND, BOARD\_WIDTH, GROUND);

drawAliens(g);

drawPlayer(g);

drawShot(g);

drawBombing(g);

}

Toolkit.getDefaultToolkit().sync();

g.dispose();

}

public void gameOver()

{

Graphics g = this.getGraphics();

g.setColor(Color.black);

g.fillRect(0, 0, BOARD\_WIDTH, BOARD\_HEIGTH);

g.setColor(new Color(0, 32, 48));

g.fillRect(50, BOARD\_WIDTH/2 - 30, BOARD\_WIDTH-100, 50);

g.setColor(Color.white);

g.drawRect(50, BOARD\_WIDTH/2 - 30, BOARD\_WIDTH-100, 50);

Font small = new Font("Helvetica", Font.BOLD, 14);

FontMetrics metr = this.getFontMetrics(small);

g.setColor(Color.white);

g.setFont(small);

g.drawString(message, (BOARD\_WIDTH - metr.stringWidth(message))/2,

BOARD\_WIDTH/2);

}

public void animationCycle() {

if (deaths == NUMBER\_OF\_ALIENS\_TO\_DESTROY) {

ingame = false;

message = "Game won!";

}

// player

player.act();

// shot

if (shot.isVisible()) {

Iterator it = aliens.iterator();

int shotX = shot.getX();

int shotY = shot.getY();

while (it.hasNext()) {

Alien alien = (Alien) it.next();

int alienX = alien.getX();

int alienY = alien.getY();

if (alien.isVisible() && shot.isVisible()) {

if (shotX >= (alienX) &&

shotX <= (alienX + ALIEN\_WIDTH) &&

shotY >= (alienY) &&

shotY <= (alienY+ALIEN\_HEIGHT) ) {

ImageIcon ii =

new ImageIcon(getClass().getResource(expl));

alien.setImage(ii.getImage());

alien.setDying(true);

deaths++;

shot.die();

}

}

}

int y = shot.getY();

y -= 4;

if (y < 0)

shot.die();

else shot.setY(y);

}

// aliens

Iterator it1 = aliens.iterator();

while (it1.hasNext()) {

Alien a1 = (Alien) it1.next();

int x = a1.getX();

if (x >= BOARD\_WIDTH - BORDER\_RIGHT && direction != -1) {

direction = -1;

Iterator i1 = aliens.iterator();

while (i1.hasNext()) {

Alien a2 = (Alien) i1.next();

a2.setY(a2.getY() + GO\_DOWN);

}

}

if (x <= BORDER\_LEFT && direction != 1) {

direction = 1;

Iterator i2 = aliens.iterator();

while (i2.hasNext()) {

Alien a = (Alien)i2.next();

a.setY(a.getY() + GO\_DOWN);

}

}

}

Iterator it = aliens.iterator();

while (it.hasNext()) {

Alien alien = (Alien) it.next();

if (alien.isVisible()) {

int y = alien.getY();

if (y > GROUND - ALIEN\_HEIGHT) {

ingame = false;

message = "Invasion!";

}

alien.act(direction);

}

}

// bombs

Iterator i3 = aliens.iterator();

Random generator = new Random();

while (i3.hasNext()) {

int shot = generator.nextInt(15);

Alien a = (Alien) i3.next();

Alien.Bomb b = a.getBomb();

if (shot == CHANCE && a.isVisible() && b.isDestroyed()) {

b.setDestroyed(false);

b.setX(a.getX());

b.setY(a.getY());

}

int bombX = b.getX();

int bombY = b.getY();

int playerX = player.getX();

int playerY = player.getY();

if (player.isVisible() && !b.isDestroyed()) {

if ( bombX >= (playerX) &&

bombX <= (playerX+PLAYER\_WIDTH) &&

bombY >= (playerY) &&

bombY <= (playerY+PLAYER\_HEIGHT) ) {

ImageIcon ii =

new ImageIcon(this.getClass().getResource(expl));

player.setImage(ii.getImage());

player.setDying(true);

b.setDestroyed(true);;

}

}

if (!b.isDestroyed()) {

b.setY(b.getY() + 1);

if (b.getY() >= GROUND - BOMB\_HEIGHT) {

b.setDestroyed(true);

}

}

}

}

public void run() {

long beforeTime, timeDiff, sleep;

beforeTime = System.currentTimeMillis();

while (ingame) {

repaint();

animationCycle();

timeDiff = System.currentTimeMillis() - beforeTime;

sleep = DELAY - timeDiff;

if (sleep < 0)

sleep = 2;

try {

Thread.sleep(sleep);

} catch (InterruptedException e) {

System.out.println("interrupted");

}

beforeTime = System.currentTimeMillis();

}

gameOver();

}

private class TAdapter extends KeyAdapter {

public void keyReleased(KeyEvent e) {

player.keyReleased(e);

}

public void keyPressed(KeyEvent e) {

player.keyPressed(e);

int x = player.getX();

int y = player.getY();

if (ingame)

{

if (e.isAltDown()) {

if (!shot.isVisible())

shot = new Shot(x, y);

}

}

}

}

}

The main logic of the game is located in the Board class.

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

for (int j=0; j < 6; j++) {

Alien alien = new Alien(alienX + 18\*j, alienY + 18\*i);

alien.setImage(ii.getImage());

aliens.add(alien);

}

}

player = new Player();

shot = new Shot();

In the gameInit() method we set up 24 aliens. The alien image size is 12x12px. We put 6px space among the aliens. We also create the player and the shot objects.

public void drawBombing(Graphics g) {

Iterator i3 = aliens.iterator();

while (i3.hasNext()) {

Alien a = (Alien) i3.next();

Alien.Bomb b = a.getBomb();

if (!b.isDestroyed()) {

g.drawImage(b.getImage(), b.getX(), b.getY(), this);

}

}

}

The drawBombing() method draws bombs launched by the aliens.

if (ingame) {

g.drawLine(0, GROUND, BOARD\_WIDTH, GROUND);

drawAliens(g);

drawPlayer(g);

drawShot(g);

drawBombing(g);

}

Inside the paint() method, we draw the ground, the aliens, the player, the shot, and the bombs.

Next we will examine the animationCycle() method.

if (deaths == NUMBER\_OF\_ALIENS\_TO\_DESTROY) {

ingame = false;

message = "Game won!";

}

If we destroy all aliens, we win the game. (24 in this game)

if (alien.isVisible() && shot.isVisible()) {

if (shotX >= (alienX) &&

shotX <= (alienX + ALIEN\_WIDTH) &&

shotY >= (alienY) &&

shotY <= (alienY+ALIEN\_HEIGHT) ) {

ImageIcon ii =

new ImageIcon(getClass().getResource(expl));

alien.setImage(ii.getImage());

alien.setDying(true);

deaths++;

shot.die();

}

}

If the shot triggered by the player collides with an alien, the alien ship is destroyed. More precisely, the dying flag is set. We use it to display an explosion. The deaths variable increases and the shot sprite is destroyed.

if (x >= BOARD\_WIDTH - BORDER\_RIGHT && direction != -1) {

direction = -1;

Iterator i1 = aliens.iterator();

while (i1.hasNext()) {

Alien a2 = (Alien) i1.next();

a2.setY(a2.getY() + GO\_DOWN);

}

}

If the aliens reach the right end of the Board, they move down and change their direction to the left.

Iterator it = aliens.iterator();

while (it.hasNext()) {

Alien alien = (Alien) it.next();

if (alien.isVisible()) {

int y = alien.getY();

if (y > GROUND - ALIEN\_HEIGHT) {

ingame = false;

message = "Invasion!";

}

alien.act(direction);

}

}

This code moves aliens. If they reach the bottom, the invasion begins.

int shot = generator.nextInt(15);

Alien a = (Alien) i3.next();

Alien.Bomb b = a.getBomb();

if (shot == CHANCE && a.isVisible() && b.isDestroyed()) {

b.setDestroyed(false);

b.setX(a.getX());

b.setY(a.getY());

}

This is the code that determines whether the alien will drop a bomb. The alien must not be destroyed. Eg. it must be visible. The bomb's destroyed flag must be set. In other words, it is alien's first bomb dropping or previous dropped bomb already hit the ground. If these two conditions are fulfilled, the bombing is left to the chance.

if (!b.isDestroyed()) {

b.setY(b.getY() + 1);

if (b.getY() >= GROUND - BOMB\_HEIGHT) {

b.setDestroyed(true);

}

}

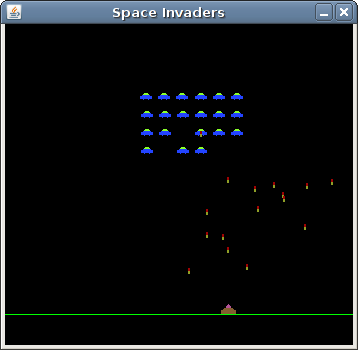
If the bomb is not destroyed, it goes 1px to the ground. If it hits the bottom, the destroyed flag is set. The alien is now ready to drop another bomb.

public void keyReleased(KeyEvent e) {

player.keyReleased(e);

}

The actual processing of this particular KeyEvent is delegated to the player sprite.

Figure: Space Invaders

This was the Space Invaders game.

# Minesweeper

In this part of the Java 2D games tutorial, we will create a Minesweeper game clone.

## 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. Further 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, there is a mine.

Board.java

package com.zetcode;

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 int[] field;

private boolean inGame;

private int mines\_left;

private Image[] img;

private int all\_cells;

private JLabel statusbar;

public Board(JLabel statusbar) {

this.statusbar = statusbar;

img = new Image[NUM\_IMAGES];

for (int i = 0; i < NUM\_IMAGES; i++) {

img[i] = (new ImageIcon(i + ".png")).getImage();

}

setDoubleBuffered(true);

addMouseListener(new MinesAdapter());

newGame();

}

private void newGame() {

Random random;

int current\_col;

int i = 0;

int position = 0;

int cell = 0;

random = new Random();

inGame = true;

mines\_left = N\_MINES;

all\_cells = N\_ROWS \* N\_COLS;

field = new int[all\_cells];

for (i = 0; i < all\_cells; i++)

field[i] = COVER\_FOR\_CELL;

statusbar.setText(Integer.toString(mines\_left));

i = 0;

while (i < N\_MINES) {

position = (int) (all\_cells \* random.nextDouble());

if ((position < all\_cells) &&

(field[position] != COVERED\_MINE\_CELL)) {

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 < all\_cells)

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 < all\_cells)

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 < all\_cells)

if (field[cell] != COVERED\_MINE\_CELL)

field[cell] += 1;

cell = position + 1;

if (cell < all\_cells)

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 < all\_cells)

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 < all\_cells)

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 < all\_cells)

if (field[cell] > MINE\_CELL) {

field[cell] -= COVER\_FOR\_CELL;

if (field[cell] == EMPTY\_CELL)

find\_empty\_cells(cell);

}

cell = j + 1;

if (cell < all\_cells)

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 cell = 0;

int uncover = 0;

for (int i = 0; i < N\_ROWS; i++) {

for (int j = 0; j < N\_COLS; j++) {

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");

}

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 rep = 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) {

rep = true;

if (field[(cRow \* N\_COLS) + cCol] <= COVERED\_MINE\_CELL) {

if (mines\_left > 0) {

field[(cRow \* N\_COLS) + cCol] += MARK\_FOR\_CELL;

mines\_left--;

statusbar.setText(Integer.toString(mines\_left));

} else

statusbar.setText("No marks left");

} else {

field[(cRow \* N\_COLS) + cCol] -= MARK\_FOR\_CELL;

mines\_left++;

statusbar.setText(Integer.toString(mines\_left));

}

}

} 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;

rep = 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 (rep)

repaint();

}

}

}

}

First we will define the constants used in our game.

private final int NUM\_IMAGES = 13;

private final int CELL\_SIZE = 15;

There are 13 images used in this game. A cell can be surrounded by maximum of 8 mines, so we need numbers 1..8. 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 15x15px.

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 N\_MINES = 40;

private final int N\_ROWS = 16;

private final int N\_COLS = 16;

The minefield in our game has 40 hidden mines. There are 16 rows and 16 columns in this field. So there are 256 cells together in the minefield.

private int[] field;

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

for (int i = 0; i < NUM\_IMAGES; i++) {

img[i] = (new ImageIcon(i + ".png")).getImage();

}

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

The newGame() initiates the Minesweeper game.

all\_cells = N\_ROWS \* N\_COLS;

field = new int[all\_cells];

for (i = 0; i < all\_cells; i++)

field[i] = COVER\_FOR\_CELL;

These lines set up the mine field. Every cell is covered by default.

i = 0;

while (i < N\_MINES) {

position = (int) (all\_cells \* random.nextDouble());

if ((position < all\_cells) &&

(field[position] != COVERED\_MINE\_CELL)) {

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 - cols;

if (cell >= 0)

if (field[cell] != COVERED\_MINE\_CELL)

field[cell] += 1;

Each of the cells can be surrounded up to 8 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 neighbor 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 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.

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.

field[(cRow \* N\_COLS) + cCol] += MARK\_FOR\_CELL;

mines\_left--;

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.

if (field[(cRow \* N\_COLS) + cCol] > COVERED\_MINE\_CELL) {

return;

}

Nothing happens if we click on the covered & 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.

Mines.java

package com.zetcode;

import java.awt.BorderLayout;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.SwingUtilities;

public class Mines extends JFrame {

private final int FRAME\_WIDTH = 250;

private final int FRAME\_HEIGHT = 290;

private final JLabel statusbar;

public Mines() {

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setSize(FRAME\_WIDTH, FRAME\_HEIGHT);

setLocationRelativeTo(null);

setTitle("Minesweeper");

statusbar = new JLabel("");

add(statusbar, BorderLayout.SOUTH);

add(new Board(statusbar));

setResizable(false);

}

public static void main(String[] args) {

SwingUtilities.invokeLater(new Runnable() {

@Override

public void run() {

JFrame ex = new Mines();

ex.setVisible(true);

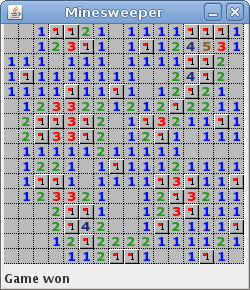
}

});

}

}

This is the main class.

Figure: Minesweeper

In this part of the Java 2D games tutorial, we created a Java clone of the Minesweeper game.

# Sokoban

In this part of the Java 2D games tutorial, we will create a Java Sokoban game clone.

## Sokoban

Sokoban is another classic computer game. It was created in 1980 by Hiroyuki Imabayashi. Sokoban means a warehouse keeper in Japanese. The player pushes boxes around a maze. The objective is to place all boxes in designated locations.

## Development

We control the sokoban object with cursor keys. We can also press the R key to restart the level. When all bags are placed on the destination areas, the game is finished. We draw "Completed" string in the left upper corner of the window.

Board.java

package sokoban;

import java.awt.Color;

import java.awt.Graphics;

import java.awt.event.KeyAdapter;

import java.awt.event.KeyEvent;

import java.util.ArrayList;

import javax.swing.JPanel;

public class Board extends JPanel {

private final int OFFSET = 30;

private final int SPACE = 20;

private final int LEFT\_COLLISION = 1;

private final int RIGHT\_COLLISION = 2;

private final int TOP\_COLLISION = 3;

private final int BOTTOM\_COLLISION = 4;

private ArrayList walls = new ArrayList();

private ArrayList baggs = new ArrayList();

private ArrayList areas = new ArrayList();

private Player soko;

private int w = 0;

private int h = 0;

private boolean completed = false;

private String level =

" ######\n"

+ " ## #\n"

+ " ##$ #\n"

+ " #### $##\n"

+ " ## $ $ #\n"

+ "#### # ## # ######\n"

+ "## # ## ##### ..#\n"

+ "## $ $ ..#\n"

+ "###### ### #@## ..#\n"

+ " ## #########\n"

+ " ########\n";

public Board() {

addKeyListener(new TAdapter());

setFocusable(true);

initWorld();

}

public int getBoardWidth() {

return this.w;

}

public int getBoardHeight() {

return this.h;

}

public final void initWorld() {

int x = OFFSET;

int y = OFFSET;

Wall wall;

Baggage b;

Area a;

for (int i = 0; i < level.length(); i++) {

char item = level.charAt(i);

if (item == '\n') {

y += SPACE;

if (this.w < x) {

this.w = x;

}

x = OFFSET;

} else if (item == '#') {

wall = new Wall(x, y);

walls.add(wall);

x += SPACE;

} else if (item == '$') {

b = new Baggage(x, y);

baggs.add(b);

x += SPACE;

} else if (item == '.') {

a = new Area(x, y);

areas.add(a);

x += SPACE;

} else if (item == '@') {

soko = new Player(x, y);

x += SPACE;

} else if (item == ' ') {

x += SPACE;

}

h = y;

}

}

public void buildWorld(Graphics g) {

g.setColor(new Color(250, 240, 170));

g.fillRect(0, 0, this.getWidth(), this.getHeight());

ArrayList world = new ArrayList();

world.addAll(walls);

world.addAll(areas);

world.addAll(baggs);

world.add(soko);

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

Actor item = (Actor) world.get(i);

if ((item instanceof Player)

|| (item instanceof Baggage)) {

g.drawImage(item.getImage(), item.x() + 2, item.y() + 2, this);

} else {

g.drawImage(item.getImage(), item.x(), item.y(), this);

}

if (completed) {

g.setColor(new Color(0, 0, 0));

g.drawString("Completed", 25, 20);

}

}

}

@Override

public void paint(Graphics g) {

super.paint(g);

buildWorld(g);

}

class TAdapter extends KeyAdapter {

@Override

public void keyPressed(KeyEvent e) {

if (completed) {

return;

}

int key = e.getKeyCode();

if (key == KeyEvent.VK\_LEFT) {

if (checkWallCollision(soko,

LEFT\_COLLISION)) {

return;

}

if (checkBagCollision(LEFT\_COLLISION)) {

return;

}

soko.move(-SPACE, 0);

} else if (key == KeyEvent.VK\_RIGHT) {

if (checkWallCollision(soko,

RIGHT\_COLLISION)) {

return;

}

if (checkBagCollision(RIGHT\_COLLISION)) {

return;

}

soko.move(SPACE, 0);

} else if (key == KeyEvent.VK\_UP) {

if (checkWallCollision(soko,

TOP\_COLLISION)) {

return;

}

if (checkBagCollision(TOP\_COLLISION)) {

return;

}

soko.move(0, -SPACE);

} else if (key == KeyEvent.VK\_DOWN) {

if (checkWallCollision(soko,

BOTTOM\_COLLISION)) {

return;

}

if (checkBagCollision(BOTTOM\_COLLISION)) {

return;

}

soko.move(0, SPACE);

} else if (key == KeyEvent.VK\_R) {

restartLevel();

}

repaint();

}

}

private boolean checkWallCollision(Actor actor, int type) {

if (type == LEFT\_COLLISION) {

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

Wall wall = (Wall) walls.get(i);

if (actor.isLeftCollision(wall)) {

return true;

}

}

return false;

} else if (type == RIGHT\_COLLISION) {

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

Wall wall = (Wall) walls.get(i);

if (actor.isRightCollision(wall)) {

return true;

}

}

return false;

} else if (type == TOP\_COLLISION) {

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

Wall wall = (Wall) walls.get(i);

if (actor.isTopCollision(wall)) {

return true;

}

}

return false;

} else if (type == BOTTOM\_COLLISION) {

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

Wall wall = (Wall) walls.get(i);

if (actor.isBottomCollision(wall)) {

return true;

}

}

return false;

}

return false;

}

private boolean checkBagCollision(int type) {

if (type == LEFT\_COLLISION) {

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

Baggage bag = (Baggage) baggs.get(i);

if (soko.isLeftCollision(bag)) {

for (int j=0; j < baggs.size(); j++) {

Baggage item = (Baggage) baggs.get(j);

if (!bag.equals(item)) {

if (bag.isLeftCollision(item)) {

return true;

}

}

if (checkWallCollision(bag,

LEFT\_COLLISION)) {

return true;

}

}

bag.move(-SPACE, 0);

isCompleted();

}

}

return false;

} else if (type == RIGHT\_COLLISION) {

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

Baggage bag = (Baggage) baggs.get(i);

if (soko.isRightCollision(bag)) {

for (int j=0; j < baggs.size(); j++) {

Baggage item = (Baggage) baggs.get(j);

if (!bag.equals(item)) {

if (bag.isRightCollision(item)) {

return true;

}

}

if (checkWallCollision(bag,

RIGHT\_COLLISION)) {

return true;

}

}

bag.move(SPACE, 0);

isCompleted();

}

}

return false;

} else if (type == TOP\_COLLISION) {

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

Baggage bag = (Baggage) baggs.get(i);

if (soko.isTopCollision(bag)) {

for (int j = 0; j < baggs.size(); j++) {

Baggage item = (Baggage) baggs.get(j);

if (!bag.equals(item)) {

if (bag.isTopCollision(item)) {

return true;

}

}

if (checkWallCollision(bag,

TOP\_COLLISION)) {

return true;

}

}

bag.move(0, -SPACE);

isCompleted();

}

}

return false;

} else if (type == BOTTOM\_COLLISION) {

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

Baggage bag = (Baggage) baggs.get(i);

if (soko.isBottomCollision(bag)) {

for (int j = 0; j < baggs.size(); j++) {

Baggage item = (Baggage) baggs.get(j);

if (!bag.equals(item)) {

if (bag.isBottomCollision(item)) {

return true;

}

}

if (checkWallCollision(bag,

BOTTOM\_COLLISION)) {

return true;

}

}

bag.move(0, SPACE);

isCompleted();

}

}

}

return false;

}

public void isCompleted() {

int num = baggs.size();

int compl = 0;

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

Baggage bag = (Baggage) baggs.get(i);

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

Area area = (Area) areas.get(j);

if (bag.x() == area.x()

&& bag.y() == area.y()) {

compl += 1;

}

}

}

if (compl == num) {

completed = true;

repaint();

}

}

public void restartLevel() {

areas.clear();

baggs.clear();

walls.clear();

initWorld();

if (completed) {

completed = false;

}

}

}

The game is simplified. It only provides the very basic functionality. The code is than easier to understand. The game has one level.

private final int OFFSET = 30;

private final int SPACE = 20;

private final int LEFT\_COLLISION = 1;

private final int RIGHT\_COLLISION = 2;

private final int TOP\_COLLISION = 3;

private final int BOTTOM\_COLLISION = 4;

The wall image size is 20x20px. This reflects the SPACE constant. The OFFSET is the distance between the borders of the window and the game world. There are four types of collisions. Each one is represented by a numerical constant.

private ArrayList walls = new ArrayList();

private ArrayList baggs = new ArrayList();

private ArrayList areas = new ArrayList();

The walls, baggs and areas are special containers, which will hold all the walls, baggs and areas of the game.

private String level =

" ######\n"

+ " ## #\n"

+ " ##$ #\n"

+ " #### $##\n"

+ " ## $ $ #\n"

+ "#### # ## # ######\n"

+ "## # ## ##### ..#\n"

+ "## $ $ ..#\n"

+ "###### ### #@## ..#\n"

+ " ## #########\n"

+ " ########\n";

This is the level of the game. Except for the space, there are five characters. The hash (#) stands for a wall. The dollar ($) represents the box to move. The dot (.) character represents the place where we must move the box. The at character (@) is the sokoban. And finally the new line character (\n) starts a new row of the world.

public final void initWorld() {

int x = OFFSET;

int y = OFFSET;

...

The initWorld() method initiates the game world. It goes through the level string and fills the above mentioned lists.

} else if (item == '$') {

b = new Baggage(x, y);

baggs.add(b);

x += SPACE;

In case of the dollar character, we create a Baggage object. The object is appended to the baggs list. The x variable is increased accordingly.

public void buildWorld(Graphics g) {

...

The buildWorld() method draws the game world on the window.

ArrayList world = new ArrayList();

world.addAll(walls);

world.addAll(areas);

world.addAll(baggs);

world.add(soko);

We create a world list which includes all objects of the game.

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

Actor item = (Actor) world.get(i);

if ((item instanceof Player)

|| (item instanceof Baggage)) {

g.drawImage(item.getImage(), item.x() + 2, item.y() + 2, this);

} else {

g.drawImage(item.getImage(), item.x(), item.y(), this);

}

...

We iterate through the world container and draw the objects. The player and the baggage images are a bit smaller. We add 2px to their coordinates to center them.

if (completed) {

g.setColor(new Color(0, 0, 0));

g.drawString("Completed", 25, 20);

}

If the level is completed, we draw "Completed" in the upper left corner of the window.

if (key == KeyEvent.VK\_LEFT) {

if (checkWallCollision(soko,

LEFT\_COLLISION)) {

return;

}

if (checkBagCollision(LEFT\_COLLISION)) {

return;

}

soko.move(-SPACE, 0);

...

Inside the keyPressed() method, we check what keys were pressed. We control the sokoban object with the cursor keys. If we press the left cursor key, we check if the sokoban collides with a wall or with a baggage. If it does not, we move the sokoban to the left.

} else if (key == KeyEvent.VK\_R) {

restartLevel();

}

We restart the level if we press the R key.

if (type == LEFT\_COLLISION) {

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

Wall wall = (Wall) walls.get(i);

if (actor.isLeftCollision(wall)) {

return true;

}

}

return false;

...

The checkWallCollision() method was created to ensure that the sokoban or a baggage do not pass the wall. There are four types of collisions. The above lines check for the left collision.

private boolean checkBagCollision(int type) {

}

The checkBagCollision() is a bit more involved. A baggage can collide with a wall, with a sokoban object or with another baggage. The baggage can be moved only if it collides with a sokoban and does not collide with another baggage or a wall. When the baggage is moved, it is time to check if the level is completed by calling the isCompleted() method.

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

Baggage bag = (Baggage) baggs.get(i);

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

Area area = (Area) areas.get(j);

if (bag.x() == area.x()

&& bag.y() == area.y()) {

compl += 1;

}

}

}

The isCompleted() method checks if the level is completed. We get the number of bags. We compare the x, y coordinates of all the bags and the destination areas.

if (compl == num) {

completed = true;

repaint();

}

The game is finished, when the completed variable equals the number of bags in the game.

public void restartLevel() {

areas.clear();

baggs.clear();

walls.clear();

initWorld();

if (completed) {

completed = false;

}

}

If we do some bad move, we can restart the level. We delete all objects from the important lists and initiate the world again. The completed variable is set to false.

Actor.java

package sokoban;

import java.awt.Image;

public class Actor {

private final int SPACE = 20;

private int x;

private int y;

private Image image;

public Actor(int x, int y) {

this.x = x;

this.y = y;

}

public Image getImage() {

return this.image;

}

public void setImage(Image img) {

image = img;

}

public int x() {

return this.x;

}

public int y() {

return this.y;

}

public void setX(int x) {

this.x = x;

}

public void setY(int y) {

this.y = y;

}

public boolean isLeftCollision(Actor actor) {

if (((this.x() - SPACE) == actor.x()) &&

(this.y() == actor.y())) {

return true;

} else {

return false;

}

}

public boolean isRightCollision(Actor actor) {

if (((this.x() + SPACE) == actor.x())

&& (this.y() == actor.y())) {

return true;

} else {

return false;

}

}

public boolean isTopCollision(Actor actor) {

if (((this.y() - SPACE) == actor.y()) &&

(this.x() == actor.x())) {

return true;

} else {

return false;

}

}

public boolean isBottomCollision(Actor actor) {

if (((this.y() + SPACE) == actor.y())

&& (this.x() == actor.x())) {

return true;

} else {

return false;

}

}

}

This is the Actor class. The class is a base class for other actors in the game. It encapsulates the basic functionality of an object in the Sokoban game.

public boolean isLeftCollision(Actor actor) {

if (((this.x() - SPACE) == actor.x()) &&

(this.y() == actor.y())) {

return true;

} else {

return false;

}

}

This method checks if the actor collides with another actor (wall, baggage, sokoban) to the left.

Wall.java

package sokoban;

import java.awt.Image;

import java.net.URL;

import javax.swing.ImageIcon;

public class Wall extends Actor {

private Image image;

public Wall(int x, int y) {

super(x, y);

URL loc = this.getClass().getResource("wall.png");

ImageIcon iia = new ImageIcon(loc);

image = iia.getImage();

this.setImage(image);

}

}

This is the Wall class. It inherits from the Actor class. Upon construction, it loads a wall image from the filesystem.

Player.java

package sokoban;

import java.awt.Image;

import java.net.URL;

import javax.swing.ImageIcon;

public class Player extends Actor {

public Player(int x, int y) {

super(x, y);

URL loc = this.getClass().getResource("sokoban.png");

ImageIcon iia = new ImageIcon(loc);

Image image = iia.getImage();

this.setImage(image);

}

public void move(int x, int y) {

int nx = this.x() + x;

int ny = this.y() + y;

this.setX(nx);

this.setY(ny);

}

}

This is the Player class. It is the class to create the sokoban object.

public void move(int x, int y) {

int nx = this.x() + x;

int ny = this.y() + y;

this.setX(nx);

this.setY(ny);

}

This class has a move() method, which moves the object inside the world.

Baggage.java

package sokoban;

import java.awt.Image;

import java.net.URL;

import javax.swing.ImageIcon;

public class Baggage extends Actor {

public Baggage(int x, int y) {

super(x, y);

URL loc = this.getClass().getResource("baggage.png");

ImageIcon iia = new ImageIcon(loc);

Image image = iia.getImage();

this.setImage(image);

}

public void move(int x, int y) {

int nx = this.x() + x;

int ny = this.y() + y;

this.setX(nx);

this.setY(ny);

}

}

This is the class for the Baggage object. This object is movable, so it has the move() method also.

Area.java

package sokoban;

import java.awt.Image;

import java.net.URL;

import javax.swing.ImageIcon;

public class Area extends Actor {

public Area(int x, int y) {

super(x, y);

URL loc = this.getClass().getResource("area.png");

ImageIcon iia = new ImageIcon(loc);

Image image = iia.getImage();

this.setImage(image);

}

}

This is the Area class. It is the object, on which we try to place the baggages.

Sokoban.java

package sokoban;

import javax.swing.JFrame;

public final class Sokoban extends JFrame {

private final int OFFSET = 30;

public Sokoban() {

InitUI();

}

public void InitUI() {

Board board = new Board();

add(board);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setSize(board.getBoardWidth() + OFFSET,

board.getBoardHeight() + 2\*OFFSET);

setLocationRelativeTo(null);

setTitle("Sokoban");

}

public static void main(String[] args) {

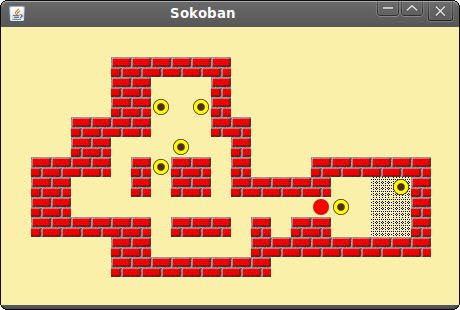
Sokoban sokoban = new Sokoban();

sokoban.setVisible(true);

}

}

This is the main class.

Figure: Sokoban

This was the Sokoban game.