**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 theactionPerformed() method. In order to use the actionPerformed() method, we must implement theActionListener 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 thepaintComponent() to be called. This way we regularly repaint the Board thus making the animation.

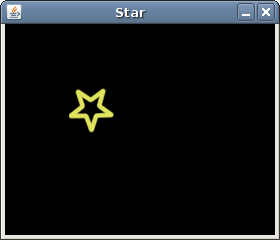


Figure: Star

**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.