Sprawozdanie nr 8

Data ćwiczeń: 07.05.2015

Data oddania sprawozdania: 18.06.2015

* Tematyka

Na ósmych zajęciach laboratoryjnych mieliśmy okazję zapoznania się z mechanizmem asynchronicznego wykonania zadań typu *Runnable*, jakim jest *Executor.* Jest to bardzo przydatna metoda, w której zamiast tworzyć osobne wątki

* Zadanie

Zadanie polegało na zaimplementowaniu programu tworzącego odpowiednie fraktale na podstawie zbiorów Mandelbrota oraz testowaniu w jakiej zależności jest czas wykonania od ilości wątków liczących. Dane do wykresu były pobrane po uruchomieniu programu na komputerze wyposażonym w procesor Intel(R) Core(TM) i7-4700MQ @ 2.40 GHz, wyposażonym w cztery rdzenie oraz umożliwiający wykorzystanie maksymalnie 8 wątków.

* Wykres
* Kod zadania

**public class** Main {  
 **public static void** main(String[] args) {  
 **long** startTime = System.*currentTimeMillis*();  
  
 **final** JFrame f = **new** JFrame(**"Mandelbrot"**);  
 f.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);  
  
 **int** width = 600;  
 **int** height = 500;  
  
 **final** Fractal fractal = **new** Mandelbrot(width, height);  
 **final** Canvas canvas = **new** FractalCanvas(fractal);  
  
  
 f.getContentPane().setPreferredSize(**new** Dimension(width, height));  
  
 f.getContentPane().add(canvas);  
 f.pack();  
 f.setLocationRelativeTo(**null**);  
  
 **final** FractalRenderer executor = **new** FractalRenderer(canvas, fractal, 600, 500);  
  
 canvas.addMouseListener(**new** MouseAdapter() {  
 @Override  
 **public void** mousePressed(MouseEvent e) {  
 **if** (e.getButton() == MouseEvent.***BUTTON1***) {  
 executor.zoomIn(e.getX(), e.getY());  
 }**else if** (e.getButton() == MouseEvent.***BUTTON3***){  
 executor.zoomOut(e.getX(), e.getY());  
  
 }  
 }  
 });  
  
 canvas.addComponentListener(**new** ComponentAdapter() {  
 @Override  
 **public void** componentResized(ComponentEvent e) {  
 **int** width = e.getComponent().getWidth();  
 **int** height = e.getComponent().getHeight();  
  
 executor.resize(width, height);  
 }  
  
 });  
  
 f.setVisible(**true**);  
  
 Thread executorThread = **new** Thread(executor);  
 executorThread.setDaemon(**true**);  
 executorThread.start();  
  
 canvas.repaint();  
 System.***out***.println(System.*currentTimeMillis*() - startTime);  
 }  
  
 **static class** FractalCanvas **extends** Canvas {  
 **private** Fractal **fractal**;  
  
 **public** FractalCanvas(Fractal fractal) {  
 **this**.**fractal** = fractal;  
 }  
  
 **public void** paint(Graphics g) {  
 g.drawImage(**fractal**.getBufferedImage(), 0, 0, Color.***red***, **null**);  
 }  
  
 @Override  
 **public void** update(Graphics g) {  
 paint(g);  
 }  
 }  
}

**public class** Gradient {  
 **private final int numColours**;  
 **private int**[][] **colourTable**;  
  
 **public** Gradient(**int** numColours, Color... colours) {  
 **this**.**numColours** = numColours;  
 generateColourTable(colours);  
 }  
  
 **private void** generateColourTable(Color... colours) {  
 **colourTable** = **new int**[**numColours**][];  
  
 **int** curColour = 0;  
 **int** gradesPerColour = **numColours** / colours.**length**;  
 **for** (**int** grad = 0; grad < colours.**length**; grad++) {  
 Color fromColour = colours[grad];  
 Color toColour = colours[(grad+1)%colours.**length**];  
 **int** startIndex = curColour \* gradesPerColour;  
 fillColour(startIndex, gradesPerColour, fromColour, toColour);  
 curColour++;  
 }  
 }  
  
 **private void** fillColour(**int** startIndex, **int** numGrades, Color fromColour, Color toColour) {  
 **for** (**int** i = startIndex; i < startIndex + numGrades; i++) {  
 **int** r = fromColour.getRed() + (**int**) ((toColour.getRed() - fromColour.getRed()) \* (i-startIndex) / numGrades);  
 **int** g = fromColour.getGreen() + (**int**) ((toColour.getGreen() - fromColour.getGreen()) \* (i-startIndex) / numGrades);  
 **int** b = fromColour.getBlue() + (**int**) ((toColour.getBlue() - fromColour.getBlue()) \* (i-startIndex) / numGrades);  
 **colourTable**[i] = **new int**[] {r, g, b};  
 }  
 }  
  
 **public int**[] getColor(**double** factor) {  
 **assert** factor >= 0 && factor < 1;  
 **return colourTable**[(**int**) (**numColours** \* factor)];  
 }  
}

**public class** Mandelbrot **implements** Fractal {  
 **private** BufferedImage **bufferedImage**;  
 **private** WritableRaster **wr**;  
  
 **private int width**;  
 **private int height**;  
  
 **private** Gradient **gradient**;  
  
 */\*\* The maximum number of iterations to use for the gradient cycle \*/* **private final int MAX\_GRAD\_ITERATIONS** = 500;  
  
 */\*\* The maximum number of threads to use for calculations \*/* **private final int THREAD\_AMOUNT** = 50;  
  
 **private double initialWidth**;  
 **private double initialHeight**;  
  
 **private double curXStart**;  
 **private double curYStart**;  
 **private long curMagnification** = 1;  
 **private volatile boolean cancelled**;  
  
 **private volatile** ExecutorService **executor**;  
  
 **public** Mandelbrot(**int** width, **int** height) {  
 setSize(width, height);  
  
 **initialWidth** = 3;  
 **initialHeight** = **initialWidth** / width \* height;  
  
 **curXStart** = -0.5-**initialWidth** / 2;  
 **curYStart** = 0-**initialHeight** / 2;  
  
 **gradient** = **new** Gradient(**MAX\_GRAD\_ITERATIONS**,  
 **new** Color(0, 0, 90), *//Navy* **new** Color(170, 255, 255), *//Light blue* **new** Color(255, 225, 50), *//Yellow* **new** Color(157, 58, 17)); *//Brown* }  
  
 **public boolean** drawFractal(**double** xPos, **double** yPos, **long** magnification, **int** iterations) {  
 **double** xSize = **initialWidth** / magnification;  
 **double** ySize = **initialHeight** / magnification;  
  
 **double** xStart = **curXStart** + (**initialWidth** / **curMagnification** \* xPos/**width**) - xSize / 2;  
 **double** yStart = **curYStart** + (**initialHeight** / **curMagnification** \* yPos/**height**) - ySize / 2;  
  
 **cancelled** = **false**;  
  
 **executor** = Executors.*newFixedThreadPool*(Runtime.*getRuntime*().availableProcessors());  
  
 **for** (**int** i = 0; i < **THREAD\_AMOUNT**; i++) {  
 **int** x = 0;  
 **int** y = i \* (**height** / **THREAD\_AMOUNT**);  
 **int** xLimit = x + **width**;  
 **int** yLimit = y + (**height** / **THREAD\_AMOUNT**);  
 **executor**.execute(**new** MandelbrotRenderer(xStart, xSize, yStart, ySize, **width**, **height**, x, y, xLimit, yLimit, iterations));  
 }  
  
 **executor**.shutdown();  
  
 **try** {  
 **executor**.awaitTermination(10, TimeUnit.***SECONDS***);  
 } **catch** (InterruptedException e) {  
 System.***err***.println(**"Rendering interrupted"**);  
 e.printStackTrace();  
 }  
  
 **curXStart** = xStart;  
 **curYStart** = yStart;  
 **curMagnification** = magnification;  
  
 **return** !**cancelled**;  
 }  
  
 **private class** MandelbrotRenderer **implements** Runnable {  
  
 **private double xStart**;  
 **private double xSize**;  
 **private double yStart**;  
 **private double ySize**;  
  
 **private int width**;  
 **private int height**;  
  
 **private int xPos**;  
 **private int yPos**;  
 **private int xLimit**;  
 **private int yLimit**;  
  
 **private int iterations**;  
  
 **public** MandelbrotRenderer(**double** xStart, **double** xSize, **double** yStart, **double** ySize,  
 **int** width, **int** height,  
 **int** x, **int** y, **int** xLimit, **int** yLimit,  
 **int** iterations) {  
  
 **this**.**xStart** = xStart;  
 **this**.**xSize** = xSize;  
 **this**.**yStart** = yStart;  
 **this**.**ySize** = ySize;  
  
 **this**.**width** = width;  
 **this**.**height** = height;  
  
 **this**.**xPos** = x;  
 **this**.**yPos** = y;  
 **this**.**yLimit** = yLimit;  
 **this**.**xLimit** = xLimit;  
  
 **this**.**iterations** = iterations;  
 }  
  
 @Override  
 **public void** run() {  
 **for** (**int** y = **yPos**; y < **yLimit**; y++) {  
 **if** (**cancelled**) **break**;  
 **for** (**int** x = **xPos**; x < **xLimit**; x++) {  
 **int** i = 0;  
 **double** xc = 0;  
 **double** yc = 0;  
  
 **double** x0 = **xStart** + **xSize** \* ((**double**)x / **width**);  
 **double** y0 = **yStart** + **ySize** \* ((**double**)y / **height**);  
  
 **while** (xc\*xc + yc\*yc < 2\*2 && i < **iterations**) {  
 **double** xtemp = xc\*xc - yc\*yc;  
 yc = 2\*xc\*yc + y0;  
 xc = xtemp + x0;  
 i++;  
 }  
  
 **wr**.setPixel(x, y, getColour(i, **iterations**));  
 }  
 }  
 }  
  
 }  
  
 **private int**[] getColour(**int** i, **int** maxIters) {  
 *//Calculate the gradient factor  
 //This should be a value between 0..1* **double** gradientFactor;  
 **if** (i == maxIters) gradientFactor = 0;  
 **else if** (maxIters < **MAX\_GRAD\_ITERATIONS**) gradientFactor = i / (**double**)maxIters;  
 **else** gradientFactor = i % **MAX\_GRAD\_ITERATIONS** / (**double**)**MAX\_GRAD\_ITERATIONS**;  
  
 **return gradient**.getColor(gradientFactor);  
 }  
  
 **public void** setSize(**int** width, **int** height) {  
 **this**.**width** = width;  
 **this**.**height** = height;  
 **bufferedImage** = **new** BufferedImage(width, height, BufferedImage.***TYPE\_INT\_RGB***);  
 **this**.**wr** = **bufferedImage**.getRaster();  
 }  
  
 **public** BufferedImage getBufferedImage() {  
 **return bufferedImage**;  
 }  
  
 **public void** cancel() {  
 **cancelled** = **true**;  
 **try** {  
 **if** (**executor** != **null**) {  
 **executor**.awaitTermination(10, TimeUnit.***SECONDS***);  
 **executor** = **null**;  
 }  
 } **catch** (InterruptedException e) {  
 System.***err***.println(**"Rendering interrupted"**);  
 e.printStackTrace();  
 }  
 }  
}

**public interface** Fractal {  
 **public boolean** drawFractal(**double** xPos, **double** yPos, **long** magnification, **int** iterations);  
 **public void** setSize(**int** width, **int** height);  
 **public** BufferedImage getBufferedImage();  
 **public void** cancel();  
}

**public class** FractalRenderer **implements** Runnable {  
 **private** Fractal **fractal**;  
 **private** Canvas **canvas**;  
 **private boolean resized** = **false**;  
 **private int width**;  
 **private int height**;  
 **private long magnification** = 1;  
 **private int iterations** = 100;  
 **public final** Object **redrawLock** = **new** Object();  
  
 **private int x**;  
 **private int y**;  
  
 **public** FractalRenderer(Canvas canvas, Fractal fractal, **int** width, **int** height) {  
 **this**.**canvas** = canvas;  
 **this**.**fractal** = fractal;  
 resize(width, height);  
 }  
  
 **public void** zoomIn(**int** x, **int** y) {  
 zoom(x, y, **magnification** \* 2, **iterations** + 75);  
 }  
  
 **public void** zoomOut(**int** x, **int** y) {  
 **if** (**magnification** > 1) {  
 zoom(x, y, **magnification** / 2, **iterations** - 75);  
 }  
 }  
  
 **private void** zoom(**int** x, **int** y, **long** magnification, **int** iterations) {  
 **this**.**x** = x;  
 **this**.**y** = y;  
 **this**.**magnification** = magnification;  
 **this**.**iterations** = iterations;  
 cancel();  
 tiggerEvent();  
 }  
  
 **private void** setSize(**int** width, **int** height) {  
 **this**.**width** = width;  
 **this**.**height** = height;  
 **x** = width / 2;  
 **y** = height / 2;  
 }  
  
 **public void** resize(**int** width, **int** height) {  
 setSize(width, height);  
 **resized** = **true**;  
 cancel();  
 tiggerEvent();  
 }  
  
 @Override  
 **public void** run() {  
 **while**(**true**) {  
 **if** (**resized**) {  
 **fractal**.setSize(**width**, **height**);  
 **resized** = **false**;  
 }  
 **synchronized** (**redrawLock**) {  
 **if** (**fractal**.drawFractal(**x**, **y**, **magnification**, **iterations**)) {  
 *//Only repaint if the fractal rendered without cancellation* **canvas**.repaint();  
 }  
 waitForEvent();  
 }  
 }  
 }  
  
 **private void** waitForEvent() {  
 **try** {  
 **synchronized**(**redrawLock**) {  
 **redrawLock**.wait();  
 }  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 **private void** tiggerEvent() {  
 **synchronized**(**redrawLock**) {  
 **redrawLock**.notify();  
 }  
 }  
  
 **public void** cancel() {  
 **fractal**.cancel();  
 }  
}