Fixed time system use

import javax.swing.\*;  
import java.awt.\*;  
import java.awt.event.\*;  
  
public class GameLoopTest extends JFrame implements ActionListener  
{  
 private GamePanel gamePanel = new GamePanel();  
 private JButton startButton = new JButton("Start");  
 private JButton quitButton = new JButton("Quit");  
 private JButton pauseButton = new JButton("Pause");  
 private boolean running = false;  
 private boolean paused = false;  
 private int fps = 60;  
 private int frameCount = 0;  
   
 public GameLoopTest()  
 {  
 super("Fixed Timestep Game Loop Test");  
 Container cp = getContentPane();  
 cp.setLayout(new BorderLayout());  
 JPanel p = new JPanel();  
 p.setLayout(new GridLayout(1,2));  
 p.add(startButton);  
 p.add(pauseButton);  
 p.add(quitButton);  
 cp.add(gamePanel, BorderLayout.CENTER);  
 cp.add(p, BorderLayout.SOUTH);  
 setSize(500, 500);  
   
 startButton.addActionListener(this);  
 quitButton.addActionListener(this);  
 pauseButton.addActionListener(this);  
 }  
   
 public static void main(String[] args)  
 {  
 GameLoopTest glt = new GameLoopTest();  
 glt.setVisible(true);  
 }  
   
 public void actionPerformed(ActionEvent e)  
 {  
 Object s = e.getSource();  
 if (s == startButton)  
 {  
 running = !running;  
 if (running)  
 {  
 startButton.setText("Stop");  
 runGameLoop();  
 }  
 else  
 {  
 startButton.setText("Start");  
 }  
 }  
 else if (s == pauseButton)  
 {  
 paused = !paused;  
 if (paused)  
 {  
 pauseButton.setText("Unpause");  
 }  
 else  
 {  
 pauseButton.setText("Pause");  
 }  
 }  
 else if (s == quitButton)  
 {  
 System.exit(0);  
 }  
 }  
   
 //Starts a new thread and runs the game loop in it.  
 public void runGameLoop()  
 {  
 Thread loop = new Thread()  
 {  
 public void run()  
 {  
 gameLoop();  
 }  
 };  
 loop.start();  
 }  
   
 //Only run this in another Thread!  
 private void gameLoop()  
 {  
 //This value would probably be stored elsewhere.  
 final double GAME\_HERTZ = 30.0;  
 //Calculate how many ns each frame should take for our target game hertz.  
 final double TIME\_BETWEEN\_UPDATES = 1000000000 / GAME\_HERTZ;  
 //At the very most we will update the game this many times before a new render.  
 //If you're worried about visual hitches more than perfect timing, set this to 1.  
 final int MAX\_UPDATES\_BEFORE\_RENDER = 5;  
 //We will need the last update time.  
 double lastUpdateTime = System.nanoTime();  
 //Store the last time we rendered.  
 double lastRenderTime = System.nanoTime();  
   
 //If we are able to get as high as this FPS, don't render again.  
 final double TARGET\_FPS = 60;  
 final double TARGET\_TIME\_BETWEEN\_RENDERS = 1000000000 / TARGET\_FPS;  
   
 //Simple way of finding FPS.  
 int lastSecondTime = (int) (lastUpdateTime / 1000000000);  
   
 while (running)  
 {  
 double now = System.nanoTime();  
 int updateCount = 0;  
   
 if (!paused)  
 {  
 //Do as many game updates as we need to, potentially playing catchup.  
 while( now - lastUpdateTime > TIME\_BETWEEN\_UPDATES && updateCount < MAX\_UPDATES\_BEFORE\_RENDER )  
 {  
 updateGame();  
 lastUpdateTime += TIME\_BETWEEN\_UPDATES;  
 updateCount++;  
 }  
   
 //If for some reason an update takes forever, we don't want to do an insane number of catchups.  
 //If you were doing some sort of game that needed to keep EXACT time, you would get rid of this.  
 if ( now - lastUpdateTime > TIME\_BETWEEN\_UPDATES)  
 {  
 lastUpdateTime = now - TIME\_BETWEEN\_UPDATES;  
 }  
   
 //Render. To do so, we need to calculate interpolation for a smooth render.  
 float interpolation = Math.min(1.0f, (float) ((now - lastUpdateTime) / TIME\_BETWEEN\_UPDATES) );  
 drawGame(interpolation);  
 lastRenderTime = now;  
   
 //Update the frames we got.  
 int thisSecond = (int) (lastUpdateTime / 1000000000);  
 if (thisSecond > lastSecondTime)  
 {  
 System.out.println("NEW SECOND " + thisSecond + " " + frameCount);  
 fps = frameCount;  
 frameCount = 0;  
 lastSecondTime = thisSecond;  
 }  
   
 //Yield until it has been at least the target time between renders. This saves the CPU from hogging.  
 while ( now - lastRenderTime < TARGET\_TIME\_BETWEEN\_RENDERS && now - lastUpdateTime < TIME\_BETWEEN\_UPDATES)  
 {  
 Thread.yield();  
   
 //This stops the app from consuming all your CPU. It makes this slightly less accurate, but is worth it.  
 //You can remove this line and it will still work (better), your CPU just climbs on certain OSes.  
 //FYI on some OS's this can cause pretty bad stuttering. Scroll down and have a look at different peoples' solutions to this.  
 try {Thread.sleep(1);} catch(Exception e) {}   
   
 now = System.nanoTime();  
 }  
 }  
 }  
 }  
   
 private void updateGame()  
 {  
 gamePanel.update();  
 }  
   
 private void drawGame(float interpolation)  
 {  
 gamePanel.setInterpolation(interpolation);  
 gamePanel.repaint();  
 }  
   
 private class GamePanel extends JPanel  
 {  
 float interpolation;  
 float ballX, ballY, lastBallX, lastBallY;  
 int ballWidth, ballHeight;  
 float ballXVel, ballYVel;  
 float ballSpeed;  
   
 int lastDrawX, lastDrawY;  
   
 public GamePanel()  
 {  
 ballX = lastBallX = 100;  
 ballY = lastBallY = 100;  
 ballWidth = 25;  
 ballHeight = 25;  
 ballSpeed = 25;  
 ballXVel = (float) Math.random() \* ballSpeed\*2 - ballSpeed;  
 ballYVel = (float) Math.random() \* ballSpeed\*2 - ballSpeed;  
 }  
   
 public void setInterpolation(float interp)  
 {  
 interpolation = interp;  
 }  
   
 public void update()  
 {  
 lastBallX = ballX;  
 lastBallY = ballY;  
   
 ballX += ballXVel;  
 ballY += ballYVel;  
   
 if (ballX + ballWidth/2 >= getWidth())  
 {  
 ballXVel \*= -1;  
 ballX = getWidth() - ballWidth/2;  
 ballYVel = (float) Math.random() \* ballSpeed\*2 - ballSpeed;  
 }  
 else if (ballX - ballWidth/2 <= 0)  
 {  
 ballXVel \*= -1;  
 ballX = ballWidth/2;  
 }  
   
 if (ballY + ballHeight/2 >= getHeight())  
 {  
 ballYVel \*= -1;  
 ballY = getHeight() - ballHeight/2;  
 ballXVel = (float) Math.random() \* ballSpeed\*2 - ballSpeed;  
 }  
 else if (ballY - ballHeight/2 <= 0)  
 {  
 ballYVel \*= -1;  
 ballY = ballHeight/2;  
 }  
 }  
   
 public void paintComponent(Graphics g)  
 {  
 //BS way of clearing out the old rectangle to save CPU.  
 g.setColor(getBackground());  
 g.fillRect(lastDrawX-1, lastDrawY-1, ballWidth+2, ballHeight+2);  
 g.fillRect(5, 0, 75, 30);  
   
 g.setColor(Color.RED);  
 int drawX = (int) ((ballX - lastBallX) \* interpolation + lastBallX - ballWidth/2);  
 int drawY = (int) ((ballY - lastBallY) \* interpolation + lastBallY - ballHeight/2);  
 g.fillOval(drawX, drawY, ballWidth, ballHeight);  
   
 lastDrawX = drawX;  
 lastDrawY = drawY;  
   
 g.setColor(Color.BLACK);  
 g.drawString("FPS: " + fps, 5, 10);  
   
 frameCount++;  
 }  
 }  
   
 private class Ball  
 {  
 float x, y, lastX, lastY;  
 int width, height;  
 float xVelocity, yVelocity;  
 float speed;  
   
 public Ball()  
 {  
 width = (int) (Math.random() \* 50 + 10);  
 height = (int) (Math.random() \* 50 + 10);  
 x = (float) (Math.random() \* (gamePanel.getWidth() - width) + width/2);  
 y = (float) (Math.random() \* (gamePanel.getHeight() - height) + height/2);  
 lastX = x;  
 lastY = y;  
 xVelocity = (float) Math.random() \* speed\*2 - speed;  
 yVelocity = (float) Math.random() \* speed\*2 - speed;  
 }  
   
 public void update()  
 {  
 lastX = x;  
 lastY = y;  
   
 x += xVelocity;  
 y += yVelocity;  
   
 if (x + width/2 >= gamePanel.getWidth())  
 {  
 xVelocity \*= -1;  
 x = gamePanel.getWidth() - width/2;  
 yVelocity = (float) Math.random() \* speed\*2 - speed;  
 }  
 else if (x - width/2 <= 0)  
 {  
 xVelocity \*= -1;  
 x = width/2;  
 }  
   
 if (y + height/2 >= gamePanel.getHeight())  
 {  
 yVelocity \*= -1;  
 y = gamePanel.getHeight() - height/2;  
 xVelocity = (float) Math.random() \* speed\*2 - speed;  
 }  
 else if (y - height/2 <= 0)  
 {  
 yVelocity \*= -1;  
 y = height/2;  
 }  
 }  
   
 public void draw(Graphics g)  
 {  
   
 }  
 }  
}

variable multi player

public void gameLoop()  
{  
 long lastLoopTime = System.nanoTime();  
 final long OPTIMAL\_TIME = 1000000000 / 60; //60fps is optimal  
   
 // keep looping round til the game ends  
 while (gameRunning)  
 {  
 // work out how long its been since the last update, this  
 // will be used to calculate how far the entities should  
 // move this loop  
 long now = System.nanoTime();  
 long updateLength = now - lastLoopTime;  
 lastLoopTime = now;  
 double delta = updateLength / ((double)OPTIMAL\_TIME);  
  
 // update the frame counter  
 lastFpsTime += updateLength;  
 fps++;  
   
 // update our FPS counter if a second has passed since  
 // we last recorded  
 if (lastFpsTime >= 1000000000)  
 {  
 System.out.println("(FPS: "+fps+")");  
 lastFpsTime = 0;  
 fps = 0;  
 }  
   
 // update the game logic  
 doGameUpdates(delta);  
   
 // draw everyting  
 render();  
   
 // we want each frame to take 10 milliseconds, to do this  
 // we've recorded when we started the frame. We add 10 milliseconds  
 // to this and then factor in the current time to give   
 // us our final value to wait for  
 // remember this is in ms, whereas our lastLoopTime etc. vars are in ns.  
 try{Thread.sleep( (lastLoopTime-System.nanoTime())/1000000 + 10 )};  
 }  
}  
  
private void doGameUpdates(double delta)  
{  
 for (int i = 0; i < stuff.size(); i++)  
 {  
 // all time-related values must be multiplied by delta!  
 Stuff s = stuff.get(i);  
 s.velocity += Gravity.VELOCITY \* delta;  
 s.position += s.velocity \* delta;  
   
 // stuff that isn't time-related doesn't care about delta...  
 if (s.velocity >= 1000)  
 {  
 s.color = Color.RED;  
 }  
 else  
 {  
 s.color = Color.BLUE;  
 }  
 }  
}