The VolatileImage API User Guide:

Managing Hardware-accelerated Offscreen Images with `VolatileImage`

Abstract

Java 2D^TM provides access to hardware-accelerated offscreen images, which take full advantage of the graphics acceleration capabilities of each Java^TM platform. The new `VolatileImage` class allows you to create and manage a hardware-accelerated offscreen image.

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

May 2001, Revision 01

Send comments about this document to:

What Is A Hardware-Accelerated Offscreen Image?

An offscreen image is stored on a surface in memory but is not currently displayed on the screen. A surface that stores such an image is called an offscreen surface.

When an application renders the offscreen image to the screen, it copies the offscreen image to the primary surface, which holds the screen contents.

Offscreen images are useful for a couple of reasons. For static images, which don’t change very often, copying an offscreen image to the screen can provide a performance advantage over re-rendering the contents every time they are needed. For example, if we use an icon in the application window, it is much more efficient to render that icon once into an image and thereafter copy that image onto the window than it would be to render the icon directly into the application window every time it was needed. A second major use of offscreen images is for double-buffering. An application can render directly into the application window on the screen, but this tends to cause flickering, especially when items in the window are animating or changing rapidly. A workaround for this artifact is to use an offscreen image as a back buffer; the application renders the contents into the back buffer and then copies the back buffer directly to the window. Since the rendering to the screen consists only of this one copy, the graphics are performed fast enough to avoid rendering artifacts.

A hardware-accelerated offscreen image is an offscreen image stored in accelerated memory, for which rendering operations can be accelerated through the graphics capabilities of the platform. For example, on Win32 systems, an image can be stored in Video RAM (VRAM) and operated on by the graphics accelerator in the PC. By using the graphics acceleration capabilities of the platform, rendering operations such as drawing to the image and copying the image to the screen or to other offscreen images can be performed without using either the system CPU or bus. So the benefits to hardware-accelerated images include both the increased speed that specialized graphics hardware can provide as well as the increased parallelism that this process exploits.

History of Offscreen Image Acceleration in the Java Platform

The Java^{TM} Development Kit (JDK) , v. 1.1, included some support for hardware acceleration of offscreen images on win32 through the use of Windows GDI functionality. Depending on the application, video card, and video card driver, GDI stored the image in video memory and used hardware-acceleration techniques for rendering to and from that image. This acceleration was unpredictable and became impossible to use with the Java 2 SDK, v 1.2.

The Java^{TM} 2, SDK, v. 1.2 included the ability to directly access the pixels of an image. Many developers requested this feature, and the new internal rendering functionality of the SDK v. 1.2 required it. For example, text is now rendered with a software renderer so that all platforms have similar text quality. However, this direct pixel access proved unworkable with the use of GDI for offscreen images because it involved copying many bits around in memory, which resulted in poor performance. For this reason, offscreen images were moved to system memory in the form of the new image type: `BufferedImage`.

The performance of `BufferedImage` software rendering has improved greatly and is acceptable for most cases that would concern users. However, there is still a need for even better performance, especially for Swing applications, which utilize double buffering. In this case, storing offscreen images in accelerated memory would allow these applications to take advantage of any hardware acceleration that the platform could provide. For example, on Windows platforms, the most basic hardware accelerators provide rectangle filling and copying speeds that both exceed the abilities of the CPU and do it in parallel with the CPU. This parallelism is an important reason to use hardware acceleration whenever possible since it frees up both the CPU and the system bus to perform other non-graphics-related functionality, thus generally improving the performance of Java applications.

The Solution: `VolatileImage`

Clearly, Java 2D^{TM} needed to restore access to hardware acceleration for offscreen images. However, the implementation used in the JDK, v. 1.1 cannot be used because it is too unpredictable and uncontrollable and is slow when performing the type of rendering required by the SDK 1.2. The other option available on win32 is DirectDraw, an API designed for giving applications direct access to video hardware memory and acceleration. DirectDraw provides control over the storage and acceleration of the image (within the limits of a given video accelerator) as well as allowing direct access to the pixels of the image.

The problem with DirectDraw is that any image stored in accelerated memory could be "lost" at any time: certain events can cause the Windows operating system to clear the image out of VRAM, thus destroying the contents of the image. These events include:

- Running another application, such as a DOS box or a game, in fullscreen mode
- Starting a screen saver
- Interrupting a task using the task manager on Win NT
- Changing screen resolution

Because of the architecture and implementation of DirectDraw, an application discovers the loss when it tries to access the image, but at that point it is too late: the contents are already gone.

With this kind of constraint on the images, DirectDraw could not be used in place of the software implementation of the SDK v. 1.2. Some event outside of the control of an application could cause an application's images to become lost, and the application might never repaint them, resulting in garbage on the screen.

New API is needed to notify an application when surface loss occurs so that the application can re-render the image. Note that it is still not possible to warn applications *before* the loss happens, but at least applications can find out about it soon afterwards and thus force a redraw of the appropriate images.

The new `VolatileImage` class allows applications to create a hardware-accelerated offscreen image and manage the contents of that image. A `VolatileImage` represents an image whose content can be lost at any time, hence the name "volatile". However, a

`VolatileImage` offers performance benefits over other kinds of images because a `VolatileImage` allows an application to take advantage of hardware acceleration, thus enabling the application to run at the native speed of the underlying platform.

In a Solaris or Linux environment, an image created with `VolatileImage` is not volatile because neither Solaris nor Linux provide a way to directly access VRAM, like DirectDraw does on win32. However, a `VolatileImage` created in a remote X environment will offer big performance gains because the offscreen image is stored in a pixmap, which resides on the server side. Instead of the X client requiring the entire image to be copied over the network to perform operations on the image, the X client only needs the instructions for performing the operations; the image itself remains on the server side.

FIGURE 1 illustrates the difference between storing and copying offscreen images using SDK 1.3 and SDK 1.4

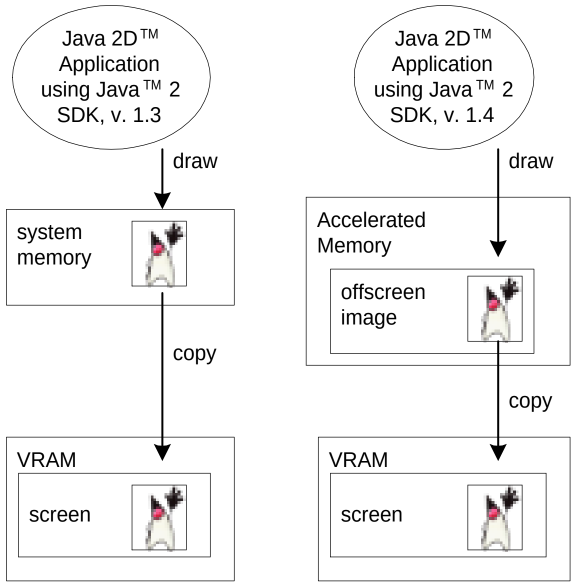
Java 2D™ Application using Java™ 2 SDK, v. 1.3
draw
system memory
copy
VRAM
screen
Java 2D™ Application using Java™ 2 SDK, v. 1.4
draw
Accelerated Memory
offscreen image
copy
VRAM
screen

FIGURE 1 Storing offscreen images with SDK 1.3 and SDK 1.4

When to Create Images With `VolatileImage`

You do not have to explicitly create a `VolatileImage` to get hardware acceleration. Java 2D also accelerates other images, but does so in a way that preserves the contents even when an event that causes surface loss occurs. Whether or not you explicitly create a `VolatileImage` depends on the needs of your application. Only constantly re-rendered images need to be explicitly created as `VolatileImage` objects to be hardware accelerated. Such images include backbuffers and animated images. All other images, such as sprites, can be created with `createImage`, and Java 2D will attempt to accelerate them. This section describes these different situations in more detail

Rendering to Surfaces with no Content Loss

If you want to render to a surface whose contents will never be lost, create the image with the `createImage(w, h)` method. This method creates an unaccelerated surface, and therefore will not experience content loss. Java 2D attempts to accelerate operations that copy from this surface by copying the image to an accelerated offscreen surface, but never does so at the risk of losing the contents.

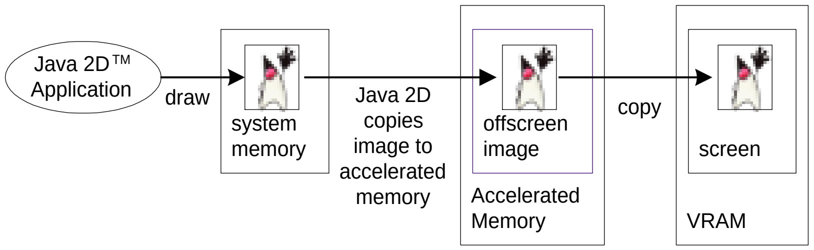

FIGURE 2 Rendering to a surface with no content loss

Rendering Dynamic Images: Double-Buffering

If you are using offscreen surfaces for double-buffering you should create the surface with `Component.createVolatileImage(w, h)` or `GraphicsConfiguration.createCompatibleVolatileImage(w, h)` because double-buffering usually involves both frequent writing to and copying from the buffer. It would be difficult or impossible for Java 2D to achieve hardware acceleration on a dynamic image that is not created as a `VolatileImage` because constant re-renderings of the image would force copies to the accelerated version from the software backup. When you use a `VolatileImage`, you should monitor the surface for content loss and re-render the surfaces appropriately during the render loops. For the SDK, version 1.4, Swing uses `VolatileImage` for its double-buffering.

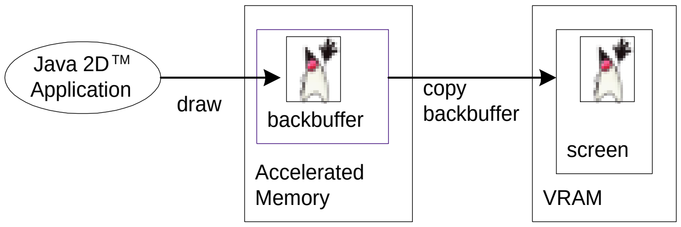

FIGURE 3 Double-Buffering

Rendering Static Images: Sprites

If you use static images, such as sprites, which are rendered to once and copied from many times, you can also use the `createImage(w, h)` method. The image is created in software memory. When the image is copied from many times, Java 2D makes a copy of the sprite in accelerated memory and future copies from the image might perform better. If the sprite's contents ever change or the accelerated version is lost, Java 2D will copy the contents of the software version again with no manual intervention from the user. To render sprites to the screen, you should use double-buffering by: creating a backbuffer with `createVolatileImage`, copying the sprite to the backbuffer, and copying the backbuffer to the primary surface. If content loss occurs, Java 2D re-copies the sprite from software memory to accelerated memory. FIGURE 4 illustrates how Java 2D accelerates copying from a sprite to a backbuffer and copying the backbuffer to the screen.

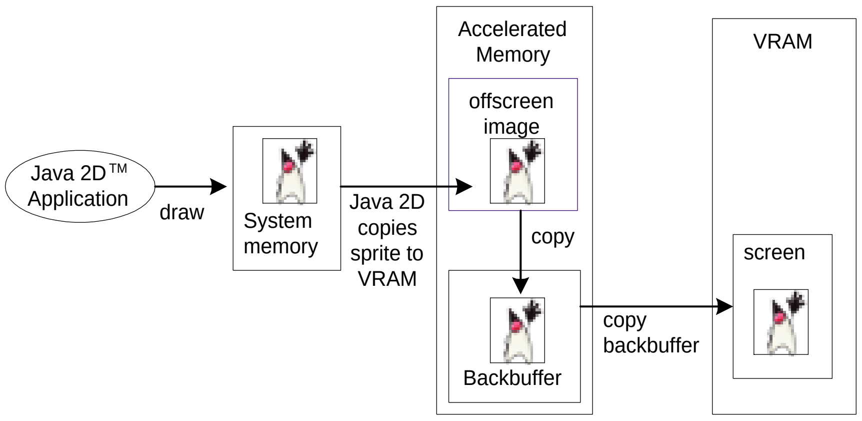

FIGURE 4 Rendering sprites

When to Use `VolatileImage` Objects on Win32

Currently, the rendering operations on win32 that are accelerated are the basic 2D graphics operations of filling and copying rectangular areas. Sometimes advanced features are also accelerated when those features can be expressed in terms of the simpler features. For example, text rendering is sometimes accelerated because the characters can be cached in accelerated memory and, when reused, copied rather than re-rendered. More complicated operations, such as diagonal lines, go through a software mechanism. This means that simple graphical applications, such as those written with Swing, use mostly hardware-accelerated features. More complex rendering features, such as curved shapes, anti-aliasing, and compositing, go through software rendering. If the bulk of your application's operations involve these complex operations then the advantage of hardware accelerated memory for your images is not as obvious: software rendering to a VRAM image tends to take longer than software rendering to a system memory image. Profile your application to see what works best for your situation.

How to Use `VolatileImage`

Using a `VolatileImage` involves these steps:

1.Create the `VolatileImage`.

2. Render to the image.

3. Test if the surface needs to be restored and restore the surface and the contents, if needed.

4. Render the image onto the target, such as the screen.

5. Test if contents are lost. If the contents are lost, return to step 3.

This example uses a `VolatileImage` for the backbuffer of an application's window:

```
VolatileImage vImg;
...
public void paint(Graphics g) {
 // Create the VolatileImage
 createBackBuffer();
 ...
 do {
     // Test if surface is lost and restore it.
     GraphicsConfiguration gc =
                this.getGraphicsConfiguration();
     int valCode = vImg.validate(gc);
     // No need to check for IMAGE_RESTORED since we are
     // to re-render the image anyway.
     if(valCode==VolatileImage.IMAGE_INCOMPATIBLE){
                createBackBuffer();
     }
     // Render to the Image
     renderFrame();
```

```
        // Render image to screen.
        g.drawImage(vImg, 0, 0, this);
        // Test if content is lost
    } while(vImg.contentsLost())
  ...

  void createBackBuffer() {
    GraphicsConfiguration.gc = getGraphicsConfiguration();
    vImg = gc.createCompatibleVolatileImage(getWidth(),
                                            getHeight());
  }
  ...
  void renderFrame() {
    // render the frame to the VolatileImage
  }
  ...
```

This example illustrates double-buffering, as explained in *When to Create Images With VolatileImage*. The VolatileDuke example in the appendix illustrates rendering sprites using double-buffering.

Managing Content Loss

Because a `VolatileImage` can lose its contents at any time, you should check for such loss before attempting to use its surface. The `VolatileImage` class has two methods that you use to test for content loss: `validate` and `contentsLost`.

The `validate` method tests two things: if the surface of the image has been lost and if the image is compatible with the current `GraphicsConfiguration`. Before trying to use a `VolatileImage`, call `validate` to check if the surface of the image has been lost. The values returned by `validate` indicate what has happened to the image since the last call to `validate`:

- If the code is `IMAGE_RESTORED` then the `validate` method has restored the surface in memory so that you can call your rendering routine to restore the image contents. If the image is being used as a backbuffer then re-rendering might not be necessary because the `validate` call might happen before the frame is rendered to that buffer. In the case of a static image, the contents must be re-rendered before the application attempts to copy from the image.
- If the code is `IMAGE_INCOMPATIBLE` then the `VolatileImage` is not compatible with the current `GraphicsConfiguration`. This incompatibility can occur if the image was created with one `GraphicsConfiguration` and then drawn into another. For example, in a multiple-monitor situation, the `VolatileImage` exists is associated with a particular `GraphicsConfiguration`. Copying that image onto a different `GraphicsConfiguration` could cause

unpredictable results. To correct this problem, you need to create a new `VolatileImage` that is compatible with the current `GraphicsConfiguration`:

```
vImg = gc.createVolatileImage(width, height);
```

- If the code is `IMAGE_OK`, nothing has happened to the image, and the application can use it.

The `validate` method takes a `GraphicsConfiguration`, which represents the graphics destination of the image. For example, it could be the `GraphicsConfiguration` associated with a particular `Component`, which the application can retrieve with `Component.getGraphicsConfiguration()`. If the image is being used as a back buffer for a `Component`, pass in the `GraphicsConfiguration` for that component. If the display environment has changed since the `VolatileImage` was last validated then it must be re-created before it can be used with this `Component`. One example of such a display environment change is if the window moved to another monitor.

A `VolatileImage` is device-specific: if you created a `VolatileImage` with one `GraphicsDevice`, you might not be able to copy that `VolatileImage` to another `GraphicsDevice`. For this reason, you need to call `validate` before attempting to copy the `VolatileImage`.

The `contentsLost` method tells the application whether the contents of the `VolatileImage` have been lost since the last call to `validate(gc)`. An application usually calls this method at the end of a loop in which all of the rendering operations associated with the image are executed. These operations include validating the image and rendering the image. If the method returns true then the loop should be processed again.

For Advanced Users: `ImageCapabilities`

The VolatileImage class also includes the `getCapabilities` method, which returns an `ImageCapabilities` object. This method is intended for the advanced user who needs to know the details of the `VolatileImage` at runtime to better select code paths in the application in different situations. Based on the return values from the `ImageCapabilities` object, the application might decide to use less images, images of lower resolution, different rendering algorithms, or various other means to attempt to get better performance from the current situation and platform.

The `ImageCapabilities` API includes two methods to test the state of the `VolatileImage` at runtime: `isAccelerated` and `isTrueVolatile`.

The `isAccelerated` method tells the application whether the image is “accelerated” or not. The definition of the term varies depending on the platform on which the application is running. For example, on win32, “accelerated” means that the image is currently located in VRAM and is able to take advantage of any hardware acceleration that DirectDraw can provide. On Solaris, which does not provide access to VRAM, “accelerated” means that the image exists in a Pixmap object on the X server, which can provide better acceleration than simple `BufferedImage` objects.

The `isTrueVolatile` method tells the application whether the image really is “volatile”, which means that the image contents can be lost at any time. On win32, this method will always return true if the image is accelerated. If the image is stored in VRAM then it can be lost at any time because of issues with the operating system or DirectDraw and is therefore considered volatile. In this case, the application should actively manage the image. On Solaris, images are never volatile: pixmaps exist as long as the application runs, and there is never a surface loss issue. The methods for managing the `VolatileImage` still function, but they basically do nothing since the image and its contents are never lost.

Performance Gains

Since `VolatileImage` takes advantage of the acceleration capabilities of the underlying platform, the performance gains that you see with `VolatileImage` depend largely on the particular graphics card that you use. The appendix includes an example called VolatileDuke that you can use to get some performance numbers. The example creates two applications. One application uses a sprite and a non-volatile backbuffer. The other application uses a sprite and a volatile backbuffer.

By running the applications with -perf, you can see how long it takes each application to run 500 iterations. When you run the example without -perf, you will see the two applications run side-by-side. This side-by-side comparison will not always show the advantages of the acceleration because the demo is highly dependent upon the threading model of the underlying platform.

Other Benefits of the VolatileImage API

Other features in the Java 2 SDK, v. 1.4 benefit from the addition of the `VolatileImage` API. These features include Swing, Full-screen Exclusive Mode API, and multibuffering.

Applications using Swing automatically get the hardware acceleration benefits of VolatileImage because Swing now uses `VolatileImage` for its double buffering.

Java applications can now use what is referred to on Windows as fullscreen/exclusive mode, which includes true back-buffer flipping and the ability to set the resolution of the monitor from inside the application. This ability allows more media-intensive applications, such as games, to access hardware-accelerated graphics functionality from within a Java application. For more information, see the Fullscreen Exclusive Mode API Tutorial.

Limitations

The development of the `VolatileImage` API is in its early stages. Below is a list of problems that are in the process of being fixed for a pending release of the SDK, v. 1.4.

- Non-rectangular fills are not accelerated.

- Anti-aliasing and alpha-blending are not accelerated.
- Text rendering to a `VolatileImage` cannot be accelerated.
- Image scaling using `VolatileImage` objects is not accelerated, but image scaling operations are generally faster in this release. For more details on image scaling performance improvements, see the High Performance Graphics whitepaper.

APPENDIX **A**

VolatileDuke.java

```
/*
 * @(#)VolatileDuke.java     1.3 01/03/05
 *
 * Copyright 2000 Sun Microsystems, Inc. All Rights Reserved.
 *
 * This software is the proprietary information of Sun Microsystems, Inc.
 * Use is subject to license terms.
 *
 */



import java.awt.*;
import java.awt.event.*;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Arc2D;
import java.awt.geom.AffineTransform;
import javax.swing.*;
import java.awt.image.VolatileImage;
import java.net.URL;




/**
 * VolatileDuke
 *
 * This test app displays 2 versions of the same applet, running
 * together.  One app uses a volatile back-buffer; the other uses
 * a non-volatile backbuffer.
 *
 * The basic applet loads the duke.gif image, copies it into a sprite
 * image, creates a back bufffer and then renders frames.  Each frame
 * rendering consists of some rendering
 * to the back buffer (coloring the background and writing some text
 * into it), copying the sprite to a different location in the back
 * buffer, and then copying the back buffer onto the screen.
 *
 * This app can also be run in "performance" mode by using the -perf
 * flag.  This mode runs the 2 apps in
 * turn at some number of iterations each.  Each test is timed and the
 * results are printed on the command line.  Running VolatileDuke in this
 * mode is the only way to get a true performance comparison of
 * the various different image configurations; due to differences in
 * thread implementations on various platforms, running both applets
 * together (as in the default case of VolatileDuke) will not necessarily
 * give the results you expect.
 *
 */



public class VolatileDuke extends JApplet implements Runnable {



    Image dukeImage;
    private static int windowW = 500, windowH = 500;
    private static int spriteW, spriteH;
    private Thread thread;
    private int xLoc = 0, yLoc = 0;
    private int xStep = 7, yStep = 3;
    Image sprite;
    Image backBuffer;
    boolean bbVolatile = false;
    static String volatileString = "Volatile";
    static String nonVolatileString = "Non-Vol";
    String bbString;
    int width, height;
```

```java
public VolatileDuke(boolean bbVolatile) {
    super();
    this.bbVolatile = bbVolatile;
    if (bbVolatile)
        bbString = volatileString;
    else
        bbString = nonVolatileString;
}



/**
 * Load the duke.gif image, create the sprite and back buffer
 * images, and render the content into the sprite.
 */
public synchronized void initOffscreen() {
    if (dukeImage == null) {
        dukeImage = new ImageIcon("duke.gif").getImage();
        spriteW = dukeImage.getWidth(null);
        spriteH = dukeImage.getHeight(null);
        sprite = createImage(spriteW, spriteH);
        restoreSpriteContent();
    }

    if (backBuffer == null ||
        width != getWidth() ||
        height != getHeight()) {
        width = getWidth();
        height = getHeight();
        if (bbVolatile)
            backBuffer = createVolatileImage(width, height);
        else
            backBuffer = createImage(width, height);
    }
}



/**
 * Prepare the sprite location variables for the next frame
 */
```

```
public void step() {
    xLoc += xStep;
    yLoc += yStep;
    if (xLoc < 0) {
        xLoc = xStep;
        xStep = -xStep;
    } else if ((xLoc + spriteW) > getWidth()) {
        xLoc = getWidth() - spriteW - xStep;
        xStep = -xStep;
    }
    if (yLoc < 0) {
        yLoc = yStep;
        yStep = -yStep;
    } else if ((yLoc + spriteH) > getHeight()) {
        yLoc = getHeight() - spriteH - yStep;
        yStep = -yStep;
    }
}



/**
 * For any of our images that are volatile, if the contents of
 * the image have been lost since the last reset, reset the image
 * and restore the contents.
 */
public void resetRestoreVolatileImages() {
    GraphicsConfiguration gc = this.getGraphicsConfiguration();
    if (bbVolatile) {
        int valCode = ((VolatileImage)backBuffer).validate(gc);
        if (valCode == VolatileImage.IMAGE_INCOMPATIBLE) {
            backBuffer = createVolatileImage(width, height);
        }
    }
}



/**
 * Renders the sprite that we will use.  We fill the sprite
 * with a background color and copy in the image that we loaded,
     */
    public void restoreSpriteContent() {
        Graphics2D g2 = (Graphics2D)sprite.getGraphics();
        g2.setColor(Color.green);
        g2.fillRect(0, 0, spriteW, spriteH);
        g2.drawImage(dukeImage, null, null);
        g2.setColor(Color.black);
        g2.dispose();
    }



    /**
     * Rendering loop of the applet.
     * We loop on rendering into the back buffer and copying
     * that buffer onto the screen.  The inner loop should usually iterate
     * only once; it will repeat if there is a problem with any of the
     * volatile surfaces.
     */
    public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        initOffscreen();



        try {
            do {
                resetRestoreVolatileImages();
                Graphics2D gBB = (Graphics2D)backBuffer.getGraphics();
                gBB.setColor(Color.green);
                gBB.fillRect(1, 1, getWidth()-1, getHeight()-1);
                gBB.setColor(Color.black);
                gBB.drawRect(0, 0, getWidth()-1, getHeight()-1);
                gBB.drawString(bbString, 5, getHeight()-1);
                gBB.drawImage(sprite, xLoc, yLoc, this);
                gBB.dispose();
                g.drawImage(backBuffer, 0, 0, this);
            } while (bbVolatile &&
                ((VolatileImage)backBuffer).contentsLost());
        } catch (Exception e) {
            System.out.println("Exception during paint: " + e);
        }
```

```
}




public void start() {
    thread = new Thread(this);
    thread.setPriority(Thread.MIN_PRIORITY);
    thread.start();
}




public synchronized void stop() {
    thread = null;
}




public void run() {
    Thread me = Thread.currentThread();
    while (thread == me) {
        step();
        Graphics g = getGraphics();
        update(g);
        g.dispose();
    }
    thread = null;
}



public long runPerf(int iterations) {
    Graphics g = getGraphics();
    long startTime = System.currentTimeMillis();
    for (int i = 0; i < iterations; ++i) {
        step();
        update(g);
    }
```

```
        Toolkit.getDefaultToolkit().sync();
        return System.currentTimeMillis() - startTime;
    }




    public static void main(String argv[]) {
        Frame f = new Frame("Java 2D(TM) Demo - VolatileDuke");
        if (argv.length > 0) {
            if (argv[0].equals("-perf"))
            {
                f.addWindowListener(new WindowAdapter() {
                    public void windowClosing(WindowEvent e) {System.exit(0);}
                });
                System.out.println("VolatileDuke Performance Test\n");
                System.out.println("     Back-Buffer     Milliseconds");
                System.out.println("     -----------     ------------");
                VolatileDuke demo = new VolatileDuke(false);
                for(int i = 0; i < 2; i++) {
                    if( i != 0 ){
                        demo = new VolatileDuke(true);
                    }
                    f.add(demo);
                    f.pack();
                    f.setSize(new Dimension(windowW, windowH));
                    f.show();
                    long perfTime = demo.runPerf(500);
                    if( i == 0 ){
                        System.out.print("  non-volatile");
                    } else {
                        System.out.print("    volatile  ");
                    }
                    System.out.println("        " + perfTime);
                    f.remove(demo);
                }
                System.exit(0);
            } else {
                System.out.println("Usage:  java VolatileDuke " +
                    "[-perf]");
                return;
            }
        } else {
            f.setLayout(new GridLayout(1, 2));
            final VolatileDuke demo1 = new VolatileDuke(false);
            f.add(demo1);
            final VolatileDuke demo2 = new VolatileDuke(true);
            f.add(demo2);
            f.addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) {System.exit(0);}
                public void windowDeiconified(WindowEvent e) {
                    demo1.start();
                    demo2.start();
                }
                public void windowIconified(WindowEvent e) {
                    demo1.stop();
                    demo2.stop();
                }
            });
            f.pack();
            f.setSize(new Dimension(windowW, windowH));
            f.show();
            demo1.start();
            demo2.start();
        }
    }
}
```