JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2008**

Vol. 7, No. 3, March-April 2008

Fixing Apples' Broken Clipboard, with Java

Douglas Lyon, Ph.D.

Abstract

The Mac OS X clipboard is infamous for changing the format of bit-mapped images that are pasted to it. These images are typically encoded using a QuickTime Tiff compressor that renders them unreadable on other platforms (e.g. Windows and Linux). This means that Mac users who create screen-shot based Word or PowerPoint documents are not able to view the images in those documents on non-Mac platforms.

QuickTime is available only under license. As a result, it is not generally available under any of the major open-source versions of UNIX (ironic, considering MacOS X is a kind of Unix). Further, windows' users (even the ones that have QuickTime) cannot decode these images (installing QuickTime is no help). As a result, the QuickTime images in documents are unreadable.

This article describes a work-around for the problem, using Java. The code has been tested and shown to work on a variety of platforms (even running under emulation using QEMU and Linux) [QEMU]. The program is distributed as a web-start application in the *JAddressBook* project.

1 THE PROBLEM

Every screen shot copied to a clipboard on a Mac is compressed using a QuickTime Tiff encoder that renders the image unreadable on multiple platforms. As a result, we seek to find a way to:

1. Capture a screen image
2. Encapsulate the image so that it is not encoded by QuickTime.
3. Paste the image into the system clipboard.
4. Remove images already on the clipboard and encapsulate them as well.

To understand the problem better, please see Figure 1.1, a mac-native screen shot.

QuickTime™ and a
TIFF (LZW) decompressor
are needed to see this picture.

Figure 1.1. A Mac-native Screen shot

Figure 1.1 shows a mac-native screen shot. Typically, the image works OK on a mac, but on a Windows computer, you get a message that reads: "QuickTime™ and a TIFF decompressor are needed to see this picture". Windows doesn't support QuickTime compression; installing QuickTime on the PC won't help.

So what do Mac users do now? They save the documents to the disk, then select "Format:insert:image-from-file" into the word/powerpoint documents. This is a bone-headed work-around for what is, in my view, an insanely broken system clipboard. The clipboard was not always broken, like this. Back in the days of *classic* the clipboard worked properly.

Logically, the Windows' user seeks to install the QuickTime product, in order to decode the TIFF compressed images. However, as of QuickTime 7.1 (Downloaded on 12/27/07) this does not help. The version is billed as being correct for both XP and Vista (however, it was not the *pro* version).

2 FIXING THE APPLE CLIPBOARD

In order to keep the Apple clipboard from automatically compressing the images using a Tiff CODEC, we must RTF (Rich Text Format) encapsulate the images. The clipboard will not compress RTF data files and MS Office products (i.e., PowerPoint and Word) can decode this format. The following code selects an area of the screen from the user, captures the screen as an image, and then sends it off to the clipboard:

```
Rectangle r = In.getRectangle("select a screen area for capture");
BufferedImage image = ImageUtils.getImage(r);
String caption = In.getString("Enter Caption");
copyImageAndTextToClipboardAsRtf(image, caption);
```

The *In* class contains a series of atomic methods that prompt the user for input.

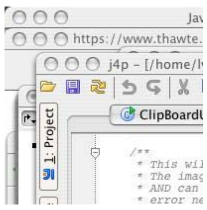

Figure 2.1. A Sample Capture

Figure 2.1 shows a sample image, captured using this technique. The *ImageUtils.getImage* method performs a screen capture using an AWT *Robot*:

```
public static BufferedImage getImage(Rectangle rect) throws AWTException {
    Robot robot = new Robot();
    return robot.createScreenCapture(rect);
  }
```

The heart of the *getImage* method is a rectangle, *rect* that is able to describe the screen area to be captured. The question of how to simulate the screen grabbing functions of a professional screen capture program remains open. There are several approaches to this problem. One is to make a screen shot of the entire desktop, show it to the user, and then draw a transparent rectangle (using rubber-banding) over the image. This approach has a lot of overhead.

For the quick and dirty approach to creating a selection rectangle (without using JNI), I display a modal dialog that obscures the screen. The rectangle is dragged into position and the user clicks "done". This is low-overhead, fast, 100% pure Java and is an ugly solution:

```
public static Rectangle getRectangle(String s) {
    final JDialog jd = new JDialog();
    jd.setTitle(s);
    Container c = jd.getContentPane();
    c.setLayout(new FlowLayout());
    jd.setSize(200, 200);
    jd.setModal(true);
    jd.setResizable(true);
    c.add(new RunButton(s) {
        public void run() {
            jd.setVisible(false);
        }
    });
    jd.setVisible(true);
    return new Rectangle(jd.getLocation(),
            jd.getSize());
}
```

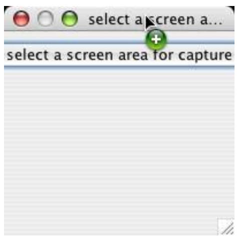

Figure 2-2. The capture rectangle

As an alternative, *JxCapture*, provides native methods on some platforms (windows) that enable access to the desktop. As a developer of a Java application, I resist the use of native methods as it complicates development and maintenance [JxCapture].

It would be really nice if the entire screen were grabbed and then drawn as a sub-rectangle into the screen capture dialog box. This would give the appearance of a transparent capture rectangle. Ideally, Sun should provide a way to better integrate Java into the desktop, so that transparent selection rectangles can be drawn. Our initial experiments in capturing sub-images from an entire screen shot showed poor update rates for our selection tool (typically greater than 1 fps, for small images using a 2.4 Ghz Intel Core Duo running MacOSX 10.4, and JDK1.5).

The final version of the screen capture program is too complex to show here. Suffice it to say that the entire screen is captured and displayed. A rubber band rectangle is used to select an area. The ROI (Region Of Interest) is then copied to the clipboard and displayed. This works at interactive speeds (better than 10 FPS on the aforementioned system).

3 BUILDING AN RTF ENCODER OF JPEG IMAGES

The RTF format is a Microsoft specification, now on version 1.9 [RTF]. It is able to encode a wide variety of different formats. There are basically two kinds of image formats, bit-mapped, and vector. Some image formats are both (i.e., hybrid formats). For example, JPEG is a bit-mapped format, but WMF can have both bit-mapped images and vector graphics. While the present example is focused on JPEG images, others are possible, given the proper encoding. The following code shows how an image and a caption can be written, as RTF to the system clipboard:

```
public static void copyImageAndTextToClipboardAsRtf(
BufferedImage image,
String caption)
        throws IOException, DocumentException {

    ByteArrayOutputStream baos =
        new ByteArrayOutputStream();
    Document doc = new Document();
    RtfWriter.getInstance(doc,baos);
    doc.open();

    byte[] bytes = ImageUtils.toJpgBytes(image);
    doc.add(new com.lowagie.text.Jpeg(bytes));


    doc.add(new Paragraph("\n" + caption));
    doc.close();
    setRtfContents(baos, null);
  }
```

Where

```
public static byte[] toJpgBytes(BufferedImage image)
        throws IOException {
    ByteArrayOutputStream outstream =
        new ByteArrayOutputStream();
    ImageIO.write(image, "jpg", outstream);
    return outstream.toByteArray();
  }
```

The RTF contents are sent to the system clipboard as a class that implements the *Transferable* interface. This is a common part of the Java clipboard framework:

```
public static void setRtfContents(
        ByteArrayOutputStream baos,
          ClipboardOwner owner) {
        ByteArrayInputStream bais =
            new ByteArrayInputStream(baos.toByteArray());
        ClipBoardUtils.setContents(new RtfTransferable(bais), owner);
    }
```

To properly identify the binary data, we declare that it is of RTF type by making use of a special string, known as the MIME type:

```
public class RtfTransferable implements Transferable {
    public DataFlavor rtfDataFlavor;
    private ByteArrayInputStream stream;
    public RtfTransferable(ByteArrayInputStream stream) {
        try {
            rtfDataFlavor =
                new DataFlavor("text/rtf; class=java.io.InputStream");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        this.stream = stream;
    }
    public DataFlavor[] getTransferDataFlavors() {
        return new DataFlavor[] { rtfDataFlavor };
    }

    public boolean isDataFlavorSupported(DataFlavor flavor) {
        return flavor.equals(rtfDataFlavor);
    }

    public Object getTransferData(DataFlavor flavor)
        throws UnsupportedFlavorException {
        if (!isDataFlavorSupported(flavor)) {
            throw new UnsupportedFlavorException(flavor);
        }
        return stream;
    }
}
```

One of the problems in getting data to the system clipboard is that the document must be converted into an RTF stream. In order to facilitate this, I make use of the *iText* framework, as described in [Lowagie]. In the following section, I show how to get an image that has already been placed on the clipboard and RTF encapsulate it.

4 RTF ENCAPSULATING AN EXISTING IMAGE.

Typically, we are faced with a clipboard that already contains an image. The image could have been placed by another program (i.e., Photoshop, or the native screen capture program). In such a case, it become necessary to RTF encapsulate the given clipboard image.

Our approach to solving this problem is to get the clipboard as an image, display it, and then paste it in, as RTF. The code follows:

```
Image image = getClipBoardImage();
String title = "clipboard";
ImageUtils.displayImage(image, title);
copyImageAndTextToClipboardAsRtf(ImageUtils.getBufferedImage(image),title);
```

Where:

```
public static Image getClipBoardImage() throws UnsupportedFlavorException,
IOException {
      Transferable t = CLIPBOARD.getContents(null);
      return (Image) t.getTransferData(DataFlavor.imageFlavor);
  }
```

5 SUMMARY

This paper demonstrates some techniques for improving the cut-and-paste situation on the Apple Macintosh. Our approach is to RTF encapsulate our images, in order to prevent the clipboard from encoding them with QuickTime codecs. Further, for images already on the clipboard, we provide a way to replace the images with RTF documents, that can be pasted into Microsoft applications (like Word and PowerPoint).

The question of what to do with data, other than image data, remains open. If the image is already embedded in an RTF document, the decoding becomes more of a challenge. There are many possible image formats available for RTF embedding (JPEG is just one of them).

After a great deal of effort in creating a screen-capture GUI, I find the native screen capture programs to be superior in look and feel. Even so, the efforts described in this paper are still useful, as the images that come from native screen capture programs still need to be converted to RTF on the clipboard, before they can be incorporated into portable office documents.

The programs described in this article have been incorporated into the *JAddressBook* program, available at: http://show.docjava.com:8086/book/cgij/code/jnlp/addbk.JAddressBook.Main.jnlp. *JAddressBook* is a Java web start application and the screen capture program is available under the *utilities* menu.

REFERENCES

[JxCapture] http://www.teamdev.com/jxcapture/demo.jsf last accessed 12/27/07.

[Lowagie] *iText in Action, Creating and Manipulating PDF* by Bruno Lowagie, Manning Publications Co. 2006, ISBN: 1932394796.

[QEMU] "QEMU is a generic and open source machine emulator and virtualizer." http://fabrice.bellard.free.fr/qemu/ last accessed 12/27/07

[RTF] "Microsoft Office Word 2007 Rich Text Format (RTF) Specification", http://www.microsoft.com/downloads/details.aspx?FamilyID=DD422B8D-FF06-4207-B476-6B5396A18A2B&displaylang=en last accessed 12/26/07.

About the author

Douglas A. Lyon (M'89-SM'00) received the Ph.D., M.S. and B.S. degrees in computer and systems engineering from Rensselaer Polytechnic Institute (1991, 1985 and 1983). Dr. Lyon has worked at AT&T Bell Laboratories at Murray Hill, NJ and the Jet Propulsion Laboratory at the California Institute of Technology, Pasadena, CA. He is currently the Chairman of the Computer Engineering Department at Fairfield University, in Fairfield CT, a senior member of the IEEE and President of DocJava, Inc., a consulting firm in Connecticut. Dr. Lyon has authored or co-authored three books (Java, Digital Signal Processing, Image Processing in Java and Java for Programmers). He has authored over 30 journal publications. Email: lyon@docjava.com. Web: http://www.DocJava.com.