Makeover Your Images With BBXImage

Back in late 2008, when the BASIS engineers were in the midst of writing the LaunchDock Utility, they frequently found themselves dealing with images in application code. Oftentimes, the source image was *almost* what they needed, but didn't quite meet the application's criteria exactly. If only there was a quick and easy way to modify the source image programmatically, either at run-time or in the form of a batch image processing utility.

Necessity, as they say, is the mother of invention and as a result, the BBXImage Utility was born. As time went by, BASIS augmented the utility with more and more capabilities, solving multiple problems as the needs arose. As its functionality grew, other BASIS utilities and demos began to rely on its capabilities and so as of BBj® 14.0, BBXImage has achieved full-blown utility status and is now installed in the **<BBjHome>/utils** directory.

What Can it Do?

The BBXImage Utility offers a wide assortment of image-related services, so it's a challenge to describe it succinctly. Generally speaking, it offers several high-level features, including:

- Retrieving an image from a variety of sources
- Modifying the image in a number of different ways
- Saving the modified image out to a file or exporting it to another program

Retrieving an initial image is rudimentary, but the utility uses the BBXImageFactory class to create a BBXImage object from a variety of different sources. The image can be a BBjImage, a Java Image, an image file on the server, or even an image from a URL (such as one from the Web).

Most of the fun comes in when modifying the image, but once that is complete, you'll need to do something with the resultant image. The utility allows you to save it as a 32-bit PNG file with an alpha channel, a JPEG file with configurable compression, or retrieve a programmatic version of the image in a BBjImage, Java BufferedImage, Java Image, or Java ImageIcon format.

By Nick Decker
Engineering Supervisor

Getting Information

Although it seems like basic functionality, we have found that often times our program needs to know the exact size of an image, but there's no easy way to get it. The utility solves this problem by providing getWidth() and getHeight() methods. In addition to being useful for the developer, the class uses this information extensively, such as when resizing the image while preserving the aspect ratio.

Image Modification

The real power of the BBXImage class is evident when it comes to modifying the image. The utility offers methods to accomplish several editing tasks, such as resizing, adding filters, rotating, flipping, cropping, rounding the corners of the image, adding a drop shadow, and more.

At BASIS, we use the BBXImage class internally for most of our screenshots to 'knock out' corners and add a border to the image. Many popular operating systems render windows with rounded corners, but image screenshot and capture utilities usually save out a rectangular image. The end result is a screenshot such as the zoomed-in version of a window shown in **Figure 1**, where the window does not have a border and the top left corner of the image shows a piece of the user's desktop behind the window's corner.

Figure 1. An enlarged top left corner of a typical screenshot with a section of the desktop visible

The BBXImage class easily solves this problem and fixes our screenshots, as we can take advantage of the roundCorners() and setBorder() methods. This removes the desktop background remnant in the corners and adds a matching border to the image. If desired, you can even add a drop shadow to make the image really stand out from its target background. The processed screenshot is shown below in **Figure 2**.

Figure 2. The same top left corner after using the BBXImage class to remove corners and add a border to the image

The utility has quickly transformed an average screenshot with distracting corner desktop remnants into a perfect specimen for documentation, presentation material, and marketing collateral. It also provides methods to round the top and bottom corners separately, so you can get the exact result you want regardless of how the original window rendered on the screen. Most of the manipulation methods, such as the roundCorners() and setBorder(), are flexible and take parameters to impact the extent of the image modification. That means that you can specify the border radius when rounding the corners, so it is possible to control how rounded the corners will be. The method also gives you the freedom to specify different sizes for the arc width and height, so the corners do not have to be circular. The screenshot in **Figure 3** shows six different examples where the border radius was set to ever-increasing sizes for the first four and the last two have different arc radii.

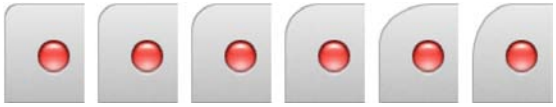

Figure 3. The effect of increasing the border radius on an image

As another example of the utility's flexibility when specifying a border for the image, you can stipulate the size, color, and opacity of the border. Adding a translucent border to an image adjusts the image's canvas size automatically, making room for the extra diameter. Saving the image as a PNG retains the opacity of the image as well as any drop shadows and borders that apply.

Other Modifications

To demonstrate just a few of the BBXImage's other manipulation capabilities, we'll start off with a simple image that we're all familiar with: the BBx® cup icon shown in **Figure 4**.

Figure 4. The original image before modification

With a few lines of code, we're able to modify the original image to suit our needs, as illustrated in **Figure 5**.

Figure 5. From left to right – the image is flipped horizontally, flipped vertically, rotated 15 degrees, has an added drop shadow, is changed to grey scale, set to 50% opacity, and darkened

The utility also offers a handy cropTransparent() method to return the smallest image possible by cropping out the transparent edges of the original. Best of all, because all of these modifications are done in a BBx program, you can transform batches of images programmatically. That makes it easy to automate the process of resizing, knocking out corners, and saving out multiple images into a target directory with a desired image format.

Resizing Images

We have seen it over and over again in movies – the detective takes a grainy traffic photo to the lab, enlarges it several times over, and ends up with a crystal clear image showing the perpetrator's license plate. Unfortunately, that's not possible in real life and just goes to show how far Hollywood is willing to stretch the truth to make a good story. In practice, resizing images is not easy and there are dozens of ways to accomplish the task. Most forms involve resampling the image, which relies on mathematical algorithms that interpolate and modify the data in the original image. When enlarging an image, many formulas exist to try to determine what the missing pixels would look like in a larger, higher resolution version of the same image. Interestingly, no one sizing method works best in all cases – each have their strengths and weaknesses. Some methods work best when enlarging images, and some are better at making images smaller. Some methods give passable results quickly, while others result in high quality images but take more time and computing power.

What BBXImage Offers for Resizing Images

The BBXImage Utility is flexible when it comes to resizing images. For starters, there are the scale() family of methods that attempt to 'do the right thing' when it comes to choosing a resampling algorithm. That means that the actual algorithm used may vary, particularly when comparing upsampling and downsampling an image. Using these methods, you can choose to scale your image by a percentage amount or provide a new absolute width and height. When you provide a scaling percentage, it retains the aspect ratio of the original image, but it can change if you provide your own width and height. Because retaining the aspect ratio is typically desired, BBXImage also offers other scaling methods where you provide the width or height and it automatically computes the new height/width, retaining the aspect ratio and scaling the image accordingly. **Figure 6** shows the result of scaling our BBj cup icon down while keeping the aspect ratio constant, as well as a version where we forced the width and height of the target image and overrode the aspect ratio.

Figure 6. A scaled version of the image with the original aspect ratio (left) and a distorted ratio (right)

Exerting Complete Control

In the few cases that you want or need control over the type of resampling to be used when scaling an image, the utility offers a scaleWithHints() method where you provide RenderingHints. RenderingHints allow you to provide input into the choice of algorithms used by Java, which performs the rendering and image manipulation services. This means that you can tell the BBXImage class to resize your image using a specific technique, such as 'Nearest Neighbor', 'Bilinear', or 'Bicubic' interpolation. To get an idea of how the specific algorithms affect the final image, look at a saved screenshot of a Dashboard chart. We used BBXImage to shrink it down to 50% of its original width and height, so the resultant image contains one fourth of the information. To get a better idea of the resampling differences, we have magnified the results as shown in **Figure 7**.

Figure 7. A comparison between scaling with Nearest Neighbor (left) and Bicubic (right) interpolations

While the 'Nearest Neighbor' results in a blockier image with a lot of information lost, the 'Bicubic' version is smoother and retains more information. For example, both images contain the bars and numbers in the chart, but the 'Nearest Neighbor' version completely eliminated the chart's borders, axis, and grid lines. Although the 'Nearest Neighbor' algorithm is very quick, it often loses information as it only samples a single pixel adjacent to any that it processes. Upscaled images using this algorithm are usually blocky and those that are scaled down often completely eliminate some of the original pixels. In contrast, the 'Bicubic' algorithm is slower as it samples the colors of the nine pixels surrounding each one it analyzes, but the final result is a better approximation of the original.

Live Demo

While the BBXImage Utility's strength lies in its ability to manipulate images programmatically, you can take it for a test drive without writing any code. Just go to links.basis.com/bbximagedemo and run a BUI program that demonstrates a few of the more commonly-used capabilities. As you can see, the utility is easy enough to use that you can spruce up your adhoc images that you include in emails to important customers, as well. A screenshot of the BUI app appears in **Figure 8**.

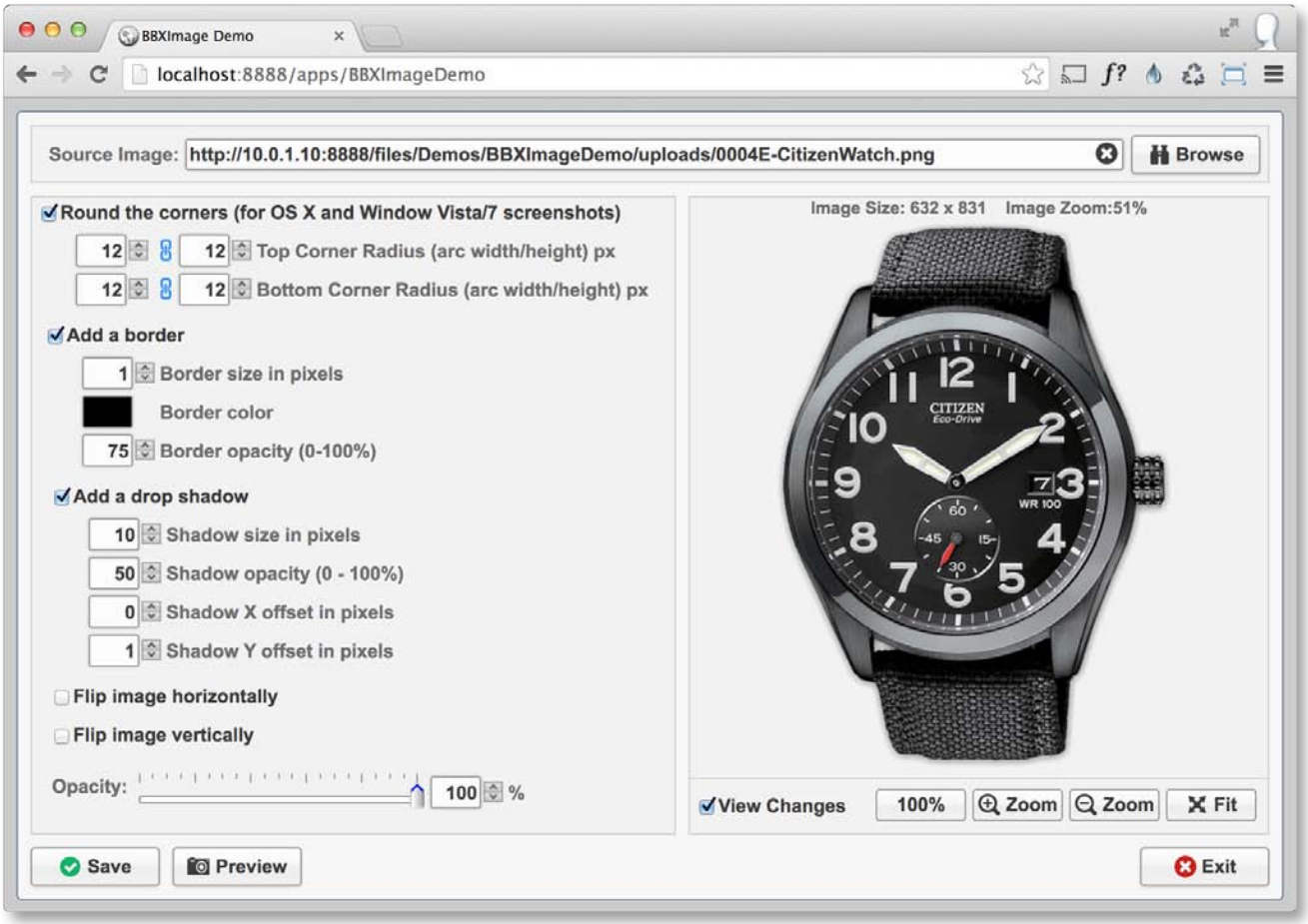

Figure 8. A BUI program demonstrates some of the BBXImage Utility's capabilities

Summary

The BBXImage Utility helps to fill a gap between BBj's graphical capabilities as offered by the BBjImageCtrl and Java's built-in image capabilities. Image processing can get deep, and typically the code required to accomplish seemingly simple tasks can be lengthy and rely on several low-level constructs. So instead of writing low-level Java code, why not use a custom BBj class that does all of the heavy lifting for you? The BBXImage Utility performs a number of common tasks so that your application images and screenshot material will look their best. Go ahead, satiate that starving artist that lives inside and make all your work look polished and professional! ■

- For more information, refer to
  - BBXImage Overview in the online Help
  - BBXImage Class JavaDocs
- Run the BUI image demonstration of the BBXImage Utility's capabilities