Looks Better, Runs Faster
GUI and BUI Image Optimizations

BASIS went to great lengths to optimize the launch time, execution, and overall performance of the new BASIS Product Suite Download page, discussed in *The Anatomy of a Web App Makeover: A Case Study* at links.basis.com/12webapp. Another area for application optimization that we explored while reviewing the new BUI download page was that of image optimization – one that developers too often woefully ignore. Image optimization is really 'low-hanging fruit' as it does not take much time to review the graphics in an app. Optimizing file sizes can lead to big speed boosts when the app runs in a high latency or tiered architecture. And choosing the best tool for the job can also ensure that images render in the best quality possible.

By Nick Decker
Engineering Supervisor

This article covers a few different ways BBj® developers can optimize their application's images to achieve quicker display times, make the most of limited bandwidth, and deliver quality output.

Format

Choosing the best image format for the individual graphics is the foremost potential optimization that developers often overlook. Various image formats, such as .png, .jpg, and .gif, employ different compression algorithms that reduce the size of the final image. These compression algorithms vary from format to format, and some are more effective for certain image types. On the flip side, some algorithms do a lousy job on images with certain characteristics, so determining the optimal format can result in huge savings. Colors are one such characteristic. A photo with millions of colors, for example, will usually compress the best using the .jpg image format. On the other hand, a logo with just a few colors will probably compress best using a .gif or .png format and won't include unsightly compression artifacts that a .jpg format could introduce.

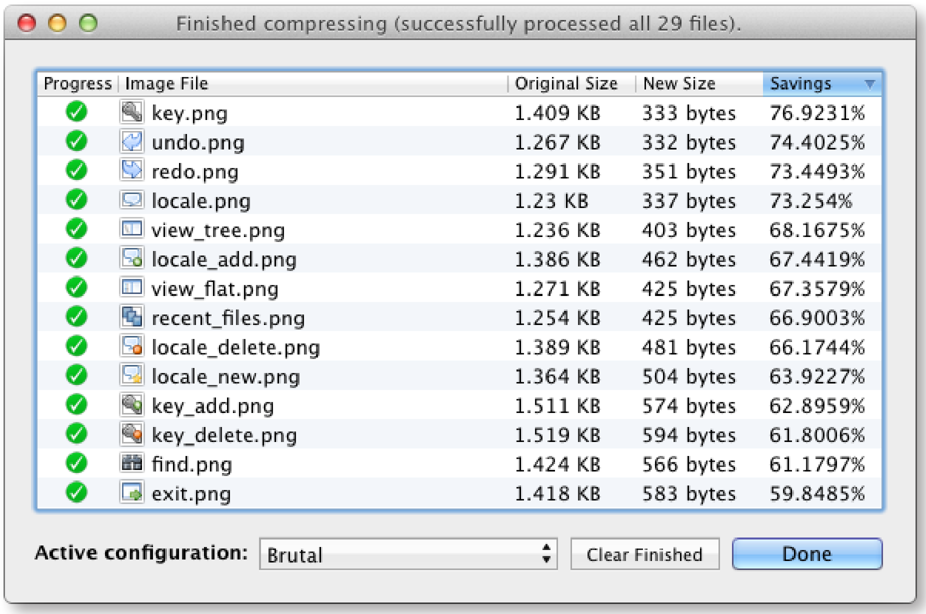
File Size

By way of example, you saved a large photograph as a .gif that weighs in at 407 KB, taking 5 seconds to download on a 1Mbps cable modem Internet connection. In addition to taking a long time to download, .gif files are palette-based and reduce the number of colors to a set of 256-fixed colors in a color table, which results in a dithered image with reduced quality. In contrast, saving the image as a .jpg file would result in a 104 KB file – almost one quarter of the size of the .gif image! Not only will this file download much more quickly, but you have the ability to control the level of compression to make a good compromise between image quality and file size.

To drive the point home, saving the exact same image as a lossless 24-bit .png image with alpha channel support (which will not even be used), would result in a 1.1MB file that takes 13 seconds to download! Selecting the optimum format for the size and type of image is occasionally a trial and error process, but it has the potential to yield huge rewards by dramatically reducing disk space and transfer time.

Crushing

Here at BASIS, we capitalized on our knowledge of 'crushing' to further optimize our BUI Mortgage demo when we changed the vertically-tiled background image from a .jpg format to an 8-bit indexed .png format. We then used a third party utility like pngcrush to strip out unnecessary metadata from the image and decreased the size of the background image by 400% – from 7.6 KB to 1.9 KB. Running several .png toolbar images used in other BASIS utilities also resulted in significant savings as shown in **Figure 1**. The new, smaller images not only take up less storage space on the disk, but more importantly, the smaller file size results in faster transfers from the server to the client. Depending on the number of images used in an application, this can have a measurable impact and reduce the application load time.


Finished compressing (successfully processed all 29 files).

Progress	Image File	Original Size	New Size	Savings
✓	key.png	1.409 KB	333 bytes	76.9231%
✓	undo.png	1.267 KB	332 bytes	74.4025%
✓	redo.png	1.291 KB	351 bytes	73.4493%
✓	locale.png	1.23 KB	337 bytes	73.254%
✓	view_tree.png	1.236 KB	403 bytes	68.1675%
✓	locale_add.png	1.386 KB	462 bytes	67.4419%
✓	view_flat.png	1.271 KB	425 bytes	67.3579%
✓	recent_files.png	1.254 KB	425 bytes	66.9003%
✓	locale_delete.png	1.389 KB	481 bytes	66.1744%
✓	locale_new.png	1.364 KB	504 bytes	63.9227%
✓	key_add.png	1.511 KB	574 bytes	62.8959%
✓	key_delete.png	1.519 KB	594 bytes	61.8006%
✓	find.png	1.424 KB	566 bytes	61.1797%
✓	exit.png	1.418 KB	583 bytes	59.8485%

Active configuration: Brutal

Figure 1. Compressing the images used by the BASIS utilities saves time and space

Uniting Images for Network Optimization

Applications vary in the number of images they employ, but it is fairly common for them to use dozens of images for menu items and toolbuttons when the application is sufficiently complex. Sending these images over the wire to the client may seem innocuous since they are usually less than 1 KB in size, but often times the number of HTTP requests required to transfer the images to the client is the crucial aspect and limiting factor. Reducing the number of HTTP requests is one of the foremost methods webmasters use to speed up their websites, making pages load faster and minimizing the effect of latency. Images are prime candidates for this form of optimization, as combining multiple discrete images into a single image file slashes the number of HTTP requests from dozens down to one. Webmasters normally rely on image maps or CSS sprites to combine image files, and BBx® developers can reap the same rewards by using an ImageList. ImageLists are typically used to set an image in a TabControl or Grid, but you can use them with any control that offers a `setImage()` method.

For example, we modified the Resource Bundle Editor to use an ImageList for the menu item, toolbutton, and button images. The application previously used 28 distinct image files, so with a little help from a CSS Sprite tool, we combined them into a single image comprised of the concatenated images as shown in **Figure 2**.

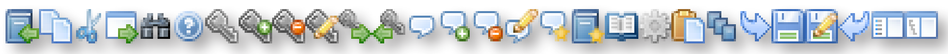


Figure 2. The result of combining multiple images into a single ImageList

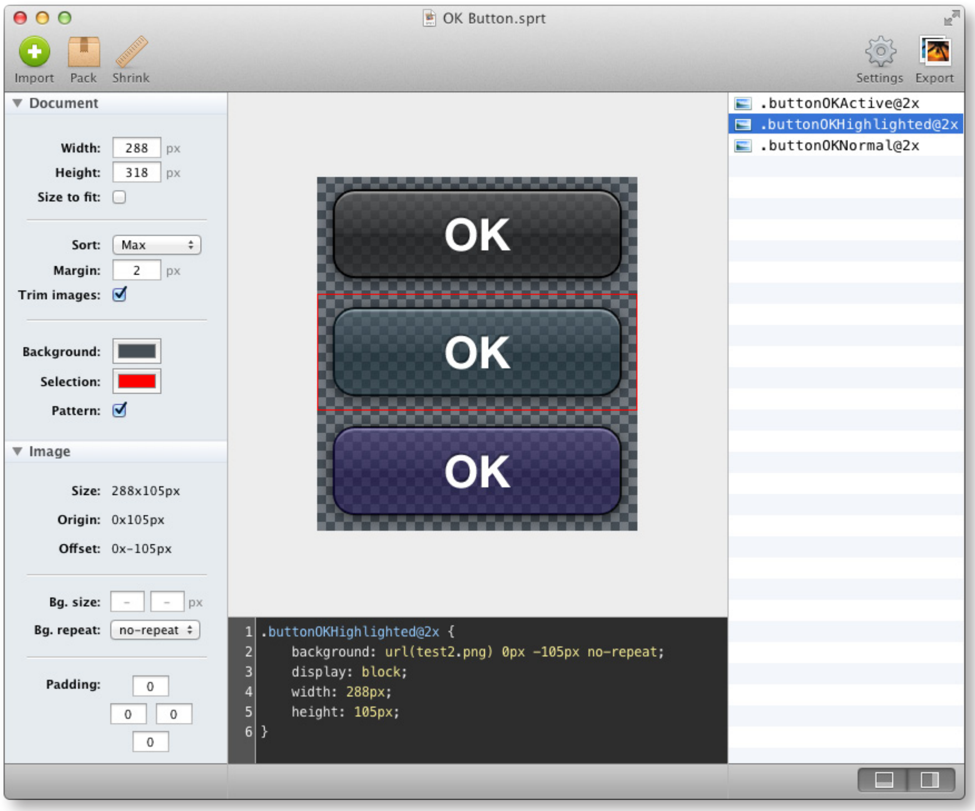
Analyzing a BUI instance of the application in Chrome's Developer Tools confirmed that the number of HTTP requests dropped from 28 down to just 1. Using the ImageList will not only drastically reduce the amount of communication between the client and server, but the resultant ImageList file will almost always compress more than the individual images (due to factors like discarding redundant header information). Our example showed that our changes not only eliminated dozens of HTTP requests, but also reduced the amount of data sent – from a combined size of 27 KB for the 28 separate images to just 10 KB for the ImageList. Combining your application's images in this fashion optimizes network performance, reduces the effect of latency, and best of all, benefits traditional thin client and BUI apps alike.

CSS Image Optimizations for BUI

If your BUI app uses multiple images as part of its CSS styling, CSS Sprites are the logical way to optimize them to reduce HTTP requests. The webpage css-tricks.com/css-sprites has a good overview of sprites with an example that shows how combining images into a single sprite reduces the number of HTTP requests from 10 down to 1 and reduces the total size of the images from 20.5 KB down to 13 KB. Even better, as images sometimes download on a 'lazy' or 'as-needed' basis, it is far better aesthetically to use a single image.

When the browser displays a control defined with separate images for the various states (normal, hovered, and active), it loads each image the first time it is needed. So loading the image files for the hovered and active states occurs when the user first interacts with the control. This takes a fraction of a second and the control flashes as the original image is removed and the new image loads and displays. When using a single sprite image, the entire image is loaded at the start and already cached by the browser when it comes time to show the other controls states so the transition occurs instantly without any visual disruption. Various free and commercial sprite editors exist, making this potentially time-consuming and exacting task a piece of cake. Some editors, such as the one shown in

Figure 3, not only take care of combining images and optimizing layout, but also write all of the CSS code for you!



Document

Width: 288 px
Height: 318 px
Size to fit: ☐

Sort: Max
Margin: 2 px
Trim images: ✓

Background: ■
Selection: 🟥
Pattern: ✓

Image

Size: 288x105px
Origin: 0x105px
Offset: 0x-105px

Bg. size: – – px
Bg. repeat: no-repeat

Padding: 0 0 0 0

.buttonOKActive@2x
.buttonOKHighlighted@2x
.buttonOKNormal@2x

```
.buttonOKHighlighted@2x {
    background: url(test2.png) 0px -105px no-repeat;
    display: block;
    width: 288px;
    height: 105px;
}
```

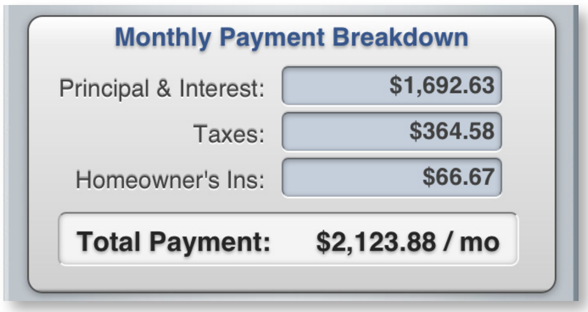
Figure 3. A sprite editor that combines images and writes the supporting CSS code

Optimizing for Retina Displays

"*Can BUI apps take advantage of one of the new 'Retina Displays'*?" is a question that has started to pop up more frequently lately. With the initial release of the iPhone 4, and with the recent releases of the latest iPad and MacBook Pros, Apple has been touting their Retina Displays. The idea is that these devices utilize high pixel density, meaning that they pack a huge amount of pixels into the device's screen. The result is a pixel density of more than 300 pixels per inch, even better than some of the early laser printers. The pixels are so small that they cannot be individually discerned when holding the device at a reasonable reading distance, meaning that text and graphics are ultra-sharp. Many Android-based phones also utilize high pixel density and their screen support API covers devices with various pixel densities and defines concepts such as density independence.

So what does this all mean to BBj BUI programmers – are their apps going to be able to play in the high-resolution game? The answer is a resounding "*Yes!*" and in most cases, programmers will not have to do anything special or make any code changes. Our aforementioned BUI Mortgage demo serves as a good example of this as seen in **Figure 4**, which displays a small section of the app running on the new iPad with its whopping 2048x1536 pixel screen resolution.

The titles, labels, and input controls are all razor-sharp and match the high-resolution native iPad apps because most BBjControls and text fall into the 'vector' category. Vector means that they can scale well without degrading in quality. However, developer supplied images used in custom CSS fall into the 'raster' category, which means that they degrade dramatically when enlarged.



Monthly Payment Breakdown

Principal & Interest:	$1,692.63
Taxes:	$364.58
Homeowner's Ins:	$66.67
Total Payment:	**$2,123.88 / mo**

Figure 4. High resolution BUI app running on a Retina Display iPad

Optimizing BUI CSS Images for Retina Displays

Optimizing custom images for a high pixel density display is possible given CSS' media query capability. The CSS in **Figure 5** specifies a background image (`my-image.png`) for regular displays. But when the client browser is using a pixel-doubled display such as the iPhone or iPad, a different image (`my-image@2x.png`) is used instead. This version of the image has four times the resolution, twice as many pixels in both the X and Y direction.

```
.my-selector {
  background-image: url(my-image.png);
  height: 100px;
  width: 200px;
}
@media only screen and (-moz-min-device-pixel-ratio: 2),
    only screen and (-o-min-device-pixel-ratio: 2/1),
    only screen and (-webkit-min-device-pixel-ratio: 2),
    only screen and (min-device-pixel-ratio: 2) {
  .my-selector {
    background-image: url(my-image@2x.png);
    background-size: 200px 100px;
  }
```

Figure 5. Defining regular and high-resolution images for Retina Displays

The **`@media`** portion of the CSS file allows developers to specify selectors that the client browser will use when viewed on a computer or mobile device with a pixel-doubled screen. The redefinition of **`.my-selector`** with the high-resolution image will take precedence over the initial definition due to CSS' cascading order. Because they both have the same weight, origin, and specificity, the last definition 'wins' and is the one that the browser will use. The final trick involves setting the **`background-size`** property for the selector to the same 200x100 pixels specified in the original definition. In essence, we are directing the browser to display the pixel-doubled image to our 'preferred' size of 200x100 pixels. If we omitted the **`background-size`** property, the image would display twice as wide and tall as the normal resolution image. By specifying the **`background-size`** property, we force it to use the native display pixels to squeeze it into the same screen real estate.

We used this same technique for the BUI Tip Calculator demo targeted for the iPhone. The app uses a custom image defined in a CSS file to display an interactive service rating as a series of stars. Simply tap on a star to define the level of service you received and the tip adjusts automatically. The "How was the service?" label is a BBjStaticText control, so it looks perfect on a pixel-doubled display without any extra work on our part. However, because the image used for the star rating system was defined as a regular image in a custom CSS file, it did not scale well and appeared blurry when viewed on the iPhone 4 and above, as shown in **Figure 6**.

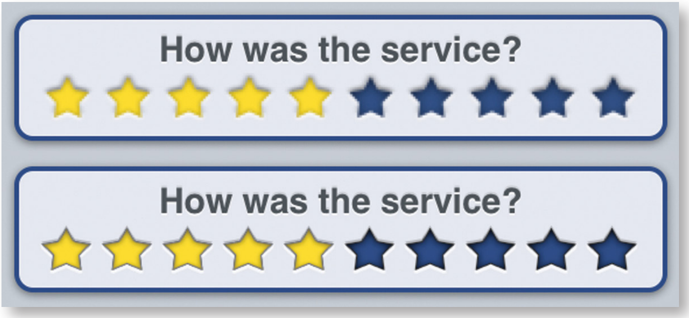


Figure 6. Comparing the non-optimized image (top) and retina-optimized image (bottom)

To make the application look great on the Retina Display iPhones, we created a version of the star image that was pixel-doubled – twice as many pixels for both the width and the height. This ensures that the image used for the rating system will be of the highest quality for every device.

Summary

As computer applications have matured and migrated from CUI to GUI and now to BUI, images have become an integral part of most of these applications. In addition to adding aesthetic value, when used wisely they improve usability and provide interactive feedback. Despite their importance, they are sometimes treated as an afterthought and even though developers typically profile their application for performance, they may overlook the importance of optimizing their application images as well.

Taking the time to analyze and compress images is an important opportunity to improve the launch speed of an application while preserving image quality. Fortunately, optimizing images is fairly easy and quick, especially with some of the advanced compression tools available. You can now accomplish this task yourself instead of delegating it to a dedicated graphic artist. BUI applications look great out-of-the-box on the new Retina and pixel-doubled displays without resorting to custom code. If your BUI app happens to utilize images in a custom CSS app, with a little CSS kung fu, you can make image degradation a distant memory and produce a fantastic looking app! ■

- Try out these BUI demos
  - Mortgage Amortization Schedule at links.basis.com/buimortgage
  - Tip Calculator at links.basis.com/buitip
  - More at links.basis.com/buidemos
- For a more in-depth discussion of image formats and criteria to help determine the best image formats, review Image File Formats at en.wikipedia.org/wiki/Image_formats
- Check out these tools
  - Pngcrush at bit.ly/4uyudU
  - Chrome Developer Tools at bit.ly/HNgdC0
  - Android Screen Support at bit.ly/mithf