Chapter 4
Image Processing and Manipulation

To make source image sets suitable for serving as tiled images, significant image processing is required. This chapter provides a discussion of the image processing techniques necessary to create a tile-based GIS. It discusses algorithms for manipulating, cutting, and scaling different types of images. Several image interpolation algorithms are given with examples and discussion of the relative benefits of each. In addition, this chapter provides guidance for choosing tile image sizes and file formats.

4.1 Basic Image Concepts

A digital image is a computer representation of a two-dimensional image and can be raster or vector based. Raster (or bitmap) digital images use a rectangular grid of picture elements (called pixels) to display the image. Vector images use geometric primitives like points, lines and polygons to represent an image. For the purposes of this book, we are dealing almost exclusively with raster images, which are composed of pixels. Chapter 11 discusses vector data in the context of tiled-mapping.

Each raster image is a grid of pixels, and each pixel represents the color of the image at that point. Typically, the individual pixels in an image are so small that they are not seen separately but blend together to form the image as seen by humans. Consider Figure 4.1; to the left is a picture of a letter A, to the right is that same picture magnified such that the individual pixels are visible.

Pixel values are expressed in units of the image's color space. A color space, or color model, is the abstract model that describes how color components can be represented. RGB (red, green, blue) is a common color model. It provides that the color components for red, green and blue be stored as separate values for each pixel. Combinations of the three values can represent many millions of visible colors. Suppose that we will use values of 0 to 1 to represent each of the components of an RGB pixel. Table 4.1 shows which combinations would create certain common colors.

J.T. Sample and E. Ioup, *Tile-Based Geospatial Information Systems: Principles and Practices*, DOI 10.1007/978-1-4419-7631-4_4,
© Springer Science+Business Media, LLC 2010

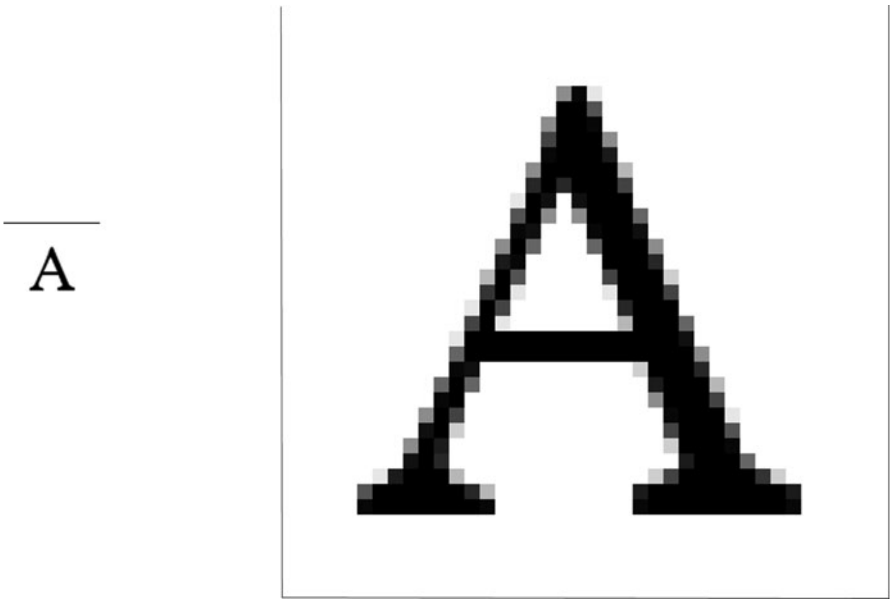


Fig. 4.1 Pixelated image.

Red	Green	Blue	Composite Color
1	0	0	Red
0	1	0	Green
0	0	1	Blue
1	1	0	Yellow
0.5	0.5	0.5	Gray
0	0	0	Black
1	1	1	White

Table 4.1 Common colors and their RGB component combination.

Systems that support transparency by means of alpha compositing add a fourth component, ranging from 0 to 1, where 0 indicates that the pixel should be fully transparent, and 1 indicates that the pixel should be fully opaque. This color model is referred to as RGBA or ARGB.

To view and manipulate digital raster images, the RGB components are most often stored as single byte values. In this case, each RGB component is an integer value from 0 to 255 instead of a real value from 0 to 1. The three components will take up 3 bytes (24 bits) or 4 bytes (32 bits) for images with alpha components. It is common to use a 4-byte integer value to store either the RGB or RGBA components.

Raster image pixels are addressable using two-dimensional coordinates with an orthogonal coordinate system over the image. We have used Cartesian coordinates for our mapping coordinate systems, where the center of the coordinates space is $(0,0)$ and horizontal or x coordinate increases as you move to the right and the vertical or y coordinate increases as you move up. Many programming environments reverse the y coordinate such that the origin of an image is at the top-left, and the y coordinate increases as you move down the image. This convention is taken from raster scan based image systems, like cathode ray tube monitors and televisions in which the top-most scanline is the first line displayed for each refresh cycle. The reversal of the y coordinate is an inconvenience that must be considered in all practical applications that relate geospatial data to digital imagery.

Raster images are stored in a variety of file formats defined mostly by their compression algorithms or lack thereof. The most commonly used formats employ compression to reduce the required disk space. Consider an example RGB image that is 1000 by 1000 pixels. To store it uncompressed would require $1000x1000x3 = 3$ megabytes. Its not unreasonable for a good image compression algorithm to obtain a 10 to 1 compression ratio. Thus, the image could be stored in 300 kilobytes.

In general there are two types of compression, lossless and lossy. Lossless algorithms compress the image's storage space without losing any information. Lossy algorithms achieve compression in part by discarding a portion of the image's information. Lossy algorithms seek to be shrewd about what portions of an image's information to discard. Many lossy algorithms can produce a compressed image which discards significant information and yet be visually identical to the original.

The most common lossy image file format is JPEG . JPEG is named after the Joint Photographic Experts Group who created the standard. There are two common lossless file formats: Portable Network Graphic (PNG), and Graphics Interchange Format (GIF). There are several common formats which do not employ compression: Bitmap (BMP), Portable Pixel Map (PPM), Portable Graymap (PGM), and Portable Bitmap (PBM).

4.2 Geospatial Images

Digital images are well suited for storage of geospatial information. This includes aerial and satellite photography, acoustic imagery, and rendered or scanned map graphics. All that is needed to make a digital image a geospatial image is to attach geospatial coordinates to the image in a manner that describes how the image covers the surface of the earth. There are two ways this is commonly done. First, you can provide the bounding rectangle for an image, as in Figure 4.2, or you can provide a single corner coordinate with the resolution of each pixel in each dimension.

Given one or more geospatial images, we can build a tile-based mapping system to distribute the data in those images.

4.2.1 Specialized File Formats

There are several file formats that have been specially adapted for storing geospatial images. MrSID (multi-resolution seamless image database) is a proprietary image

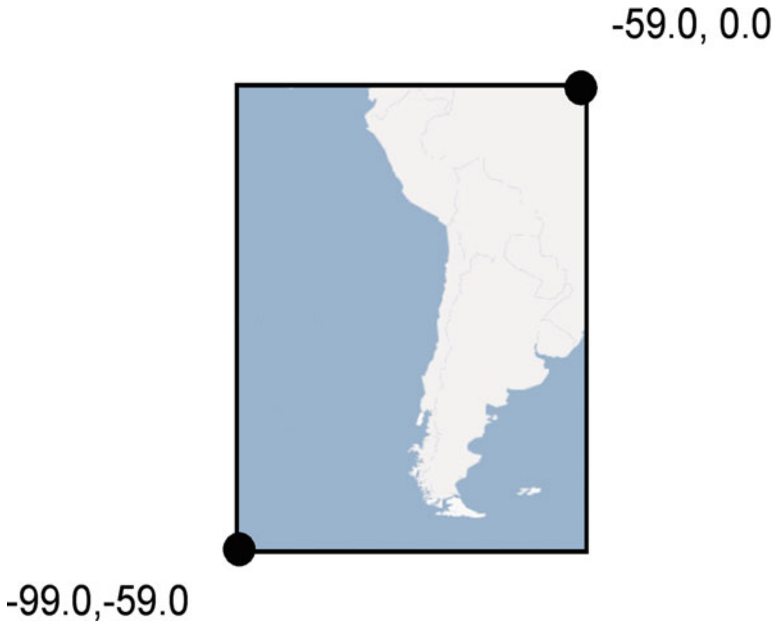


Fig. 4.2 Example geospatial image with bounding rectangle defined.

storage format produced by LizardTech. It is specially designed for storage of large geospatial images, most commonly ortho-rectified imagery. MrSID uses a wavelet based compression to store multiple resolutions of the image. This allows for fast access to overview (or thumbnail) sections of the image. It is not uncommon for MrSID images to be generated with many millions of pixels.

JPEG2000 is the next generation file format produced by the Joint Photographic Experts Group. Like MrSID, it is a wavelet based format. JPEG2000 was not specially designed to store geospatial imagery; however, common extensions have been made that allow geospatial information to be attached to the images. JPEG2000 is also well suited to storage of very large images and is a more open format than MrSID.

One of the oldest and most common geospatial image file format is GEOTIFF. GEOTIFF is based on the Tagged Image File Format (TIFF) standard. A GEOTIFF is simply a TIFF file with standard geospatial tags added to it. The TIFF standard is, perhaps, the broadest of any common image file format. It allows many options including alternate compression schemes or no compression at all. It also allows for multi-page images, a variety of color models, and a variety of storage layouts. Fortunately, there are open source software packages for reading and writing TIFF (and GEOTIFF) files that simplify the task of dealing with this complicated image format.

It should be noted that in some cases, geospatial imagery will be stored in files that do not support embedded geospatial coordinates. In those cases, it is customary to provide an accompanying file with the coordinates in it. This is only a convention, not a formal standard. Therefore, the technical details will vary from one implementation to another.

4.3 Image Manipulation

This section will provide background on the image manipulation algorithms needed for the tile creation process, which will be covered in the next chapter. Recall that tiled images are stored in fixed resolutions. It is highly unlikely that a collection of source images will match any single fixed resolution. Since we use multiple resolutions, even if our source images match one resolution, it's impossible for them to match all of our resolutions. Therefore, we are going to have to perform some image scaling.

Image scaling is a type of interpolation. Interpolation is the process of creating new data values within the range of a discrete set of known data values. We will first examine the basic algorithm for scaling and subsetting images. Then we will explain three common interpolation algorithms:

- Nearest Neighbor
- Bilinear
- Bicubic

Each interpolation algorithm has different characteristics with respect to computational performance and output image quality. For the algorithms provided below, we assume that our images have a single color channel. This simplifies the explanation of the techniques. To use the algorithms with three channel color images, the steps are simply repeated for each channel.

The basic component of all of our image scaling algorithms is the same. We will construct a target image, t, and then iterate over the pixels in t, filling them in with data computed from the pixels in our source images. Each image is treated as a two-dimensional array.

The following are some common definitions that will be used in all our image scaling algorithms (see Listing 4.1 and Figures 4.3 and 4.4). The image scaling algorithms will reference a generic interpolation function ”interpolate” (Listing 4.2). The first parameters are the details of the source image. ”tx” and ”ty” are the map coordinates of a pixel that is to be interpolated from the source data. The image scaling algorithms also reference a common function ”geolocate” that calculates the geographical coordinate of the center of a pixel (Listing 4.3). Because images are stored in scanline order, the y coordinates have to be flipped. The variable `adj_j` is created to do this.

In the first scaling algorithm (Listing 4.4 and Figure 4.5), we make a simplification assumption that the source image and the target image have the same map coor-

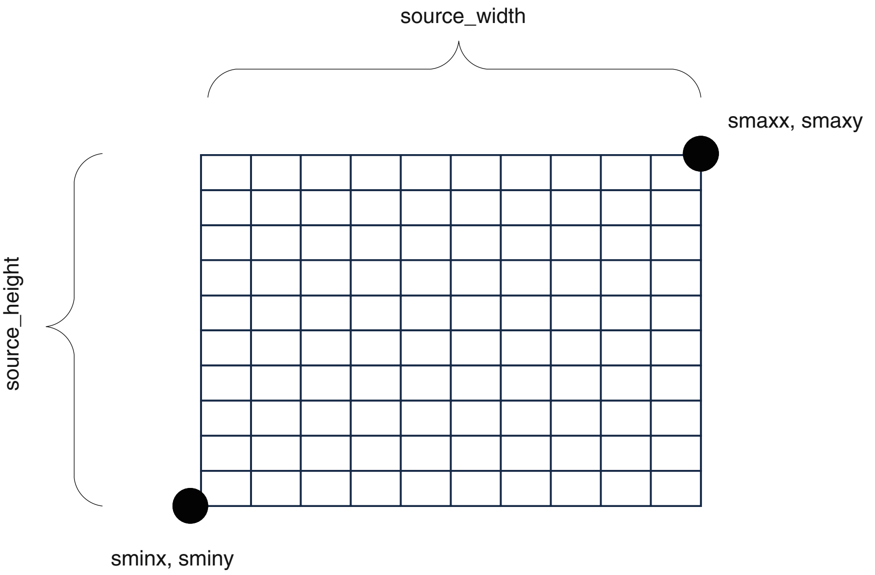


Fig. 4.3 Source image parameters: `sminx`, `sminy`, `smaxx`, `smaxy`, `source_width`, and `source_height`.

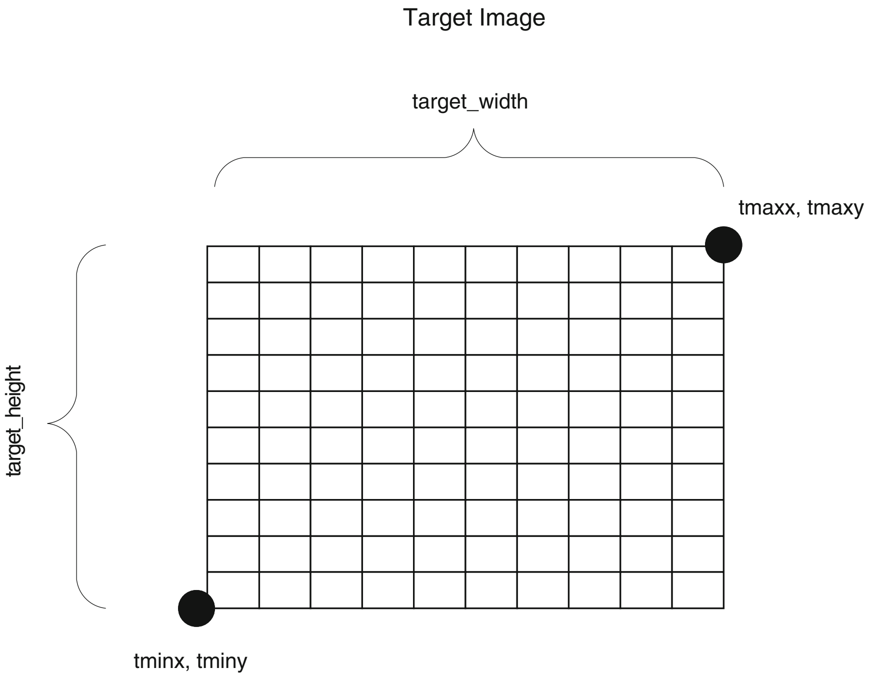


Fig. 4.4 Target image and parameters: `tminx`, `tminy`, `tmaxx`, `tmaxy`, `target_width` and `target_height`.

Listing 4.1 Definitions of variables in code examples.

```
integer s[][]:  Source Image, a 2-d array
integer source_width: width of source image
integer source_height: height of source image
real sminx: minimum horizontal map coordinate of source image
real sminy: minimum vertical map coordinate of source image
real smaxx: maximum horizontal map coordinate of source image
real smaxy: maximum vertical map coordinate of source image

integer t[][]:  Target Image, a 2-d array
integer target_width: width of target image
integer target_height: height of target image
real tminx: minimum horizontal map coordinate of target image
real tminy: minimum vertical map coordinate of target image
real tmaxx: maximum horizontal map coordinate of target image
real tmaxy: maximum vertical map coordinate of target image
```

Listing 4.2 Definition of abstract function interpolate that will be implemented by specific algorithms.

```
function integer interpolate(s, sminx, sminy, smaxx, smaxy, source_width,
     source_height, tx, ty)
```

Listing 4.3 Compute the geographic coordinates of the center of a pixel.

```
function real,real geolocate(real minx,miny,maxx,maxy, integer i,j,width,height
    )
    comment:
        minx,miny,maxx,maxy are the geographical coordinates of the image
        width and height are the dimensions of the image
        i and j are the pixel coordinates to be converted to geographic
             coordinates

    real pixel_width= (maxx-minx) / width
    real pixel_height= (maxy-miny) / height

    real x =(i + 0.5) * pixel_width  + minx

    int adj_j = height - j - 1

    real y =(adj_j + 0.5) * pixel_height + miny

    comment: we offset by 0.5 the indexes, to get the center of the pixel

    return x,y
```

dinates but different dimensions. So, `sminx` = `tminx`, `miny` = `tminy`, `smaxx` = `tmaxx`, and `smaxy` = `ymaxy`, but `source_width` ≠ `target_width` and `source_height` ≠ `target_height`.

Care should be taken in determining whether to iterate in row-major or column-major order. This is a practical consideration that must be made in the context of specific programming environments. Java, C, and many others store array data in row-major format . Iterating in this fashion can potentially greatly improve the performance of the algorithm due to the principle of Locality of Reference. In the con-

Listing 4.4 Simple Image Scaling: source and target images have the same geographic coordinates but different sizes.

```
for j in xrange(target_height):
    for i in xrange(target_width):
        (tx, ty) = geolocate(tminx, tminy, tmaxx, tmaxy, i, j, target_width,
             target_height)
        val = interpolate(s, sminx, sminy, smaxx, smaxy, source_width,
             source_height, tx, ty)
        t[j][i] = val
```

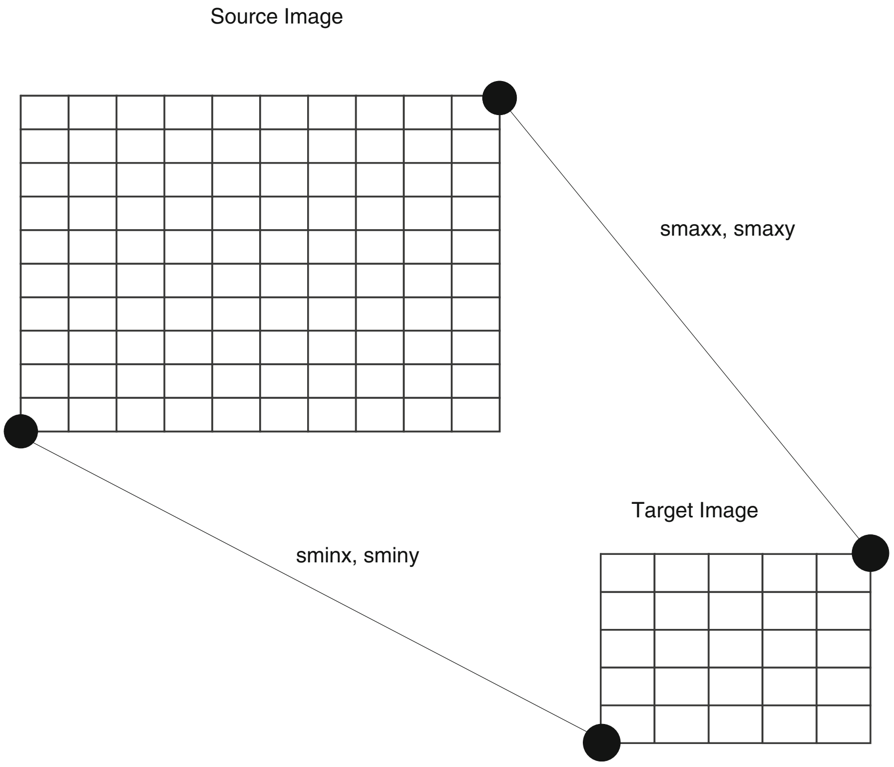


Fig. 4.5 In Scaling Algorithm 1, the source and target images share coordinates.

text of digital image manipulation, it means we should access pixels in roughly the order they are stored in the computer's memory. This reduces the number of times the operating system has to pull new memory pages into the cache [1]. Our second image scaling algorithm (Listing 4.5 and Figure 4.6) is a more general version of algorithm 1. In it, t is a scaled subsection of s. This algorithm is suitable as a basis for almost any rescaling and subsetting task.

Next, we will define our interpolation algorithms. Each of following interpolation algorithms implements the generic "interpolate" function defined earlier. In general, interpolation solves the problem shown in Figure 4.7. That is, we want to get the

Listing 4.5 Target image is a scaled subsection of source image.

```
for j = 0 to target_height - 1,
    for i = 0 to target_width - 1,
        real tx;{tx is the target pixel's x coordinate}
        real ty;{ty is the target pixel's y coordinate}
        tx,ty = geolocate(tminx,tminy,tmaxx,
                    tmaxy,i,j,target_width ,target_height)

        integer pixel_val=  interpolate(s, sminx, sminy, smaxx, smaxy,
             source_width, source_height, tx, ty);

        t[j][i] = pixel_val;
    end if
end if
```

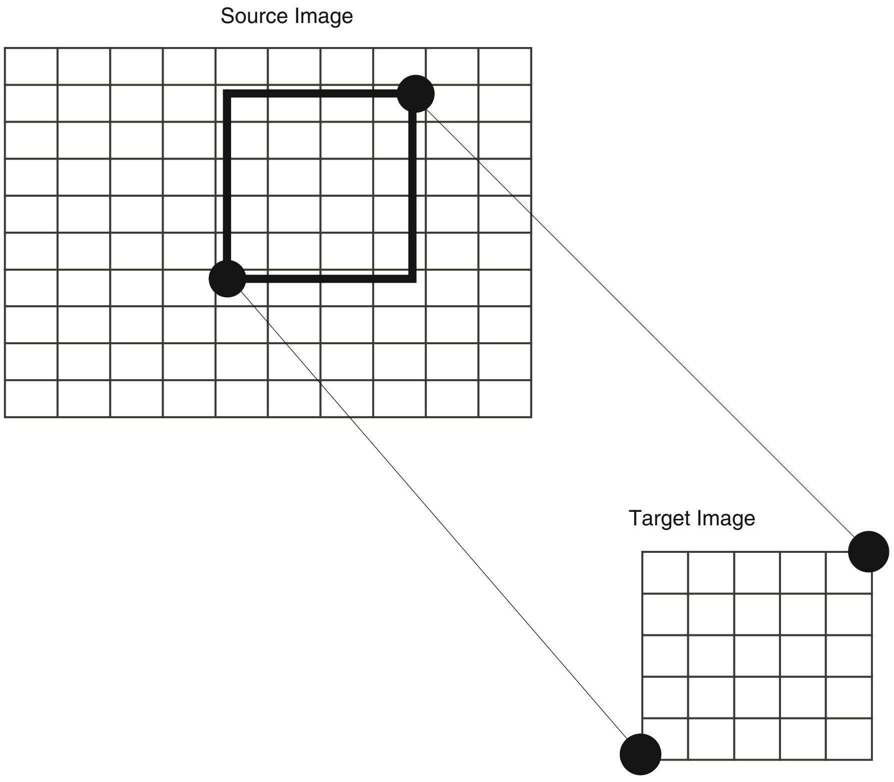


Fig. 4.6 The target image is a scaled subset of the source image.

value for a target pixel that does not correlate exactly to a source pixel. In this case, the target pixel overlaps pixels (2,1) , (3,1), (2,2), and (3,2).

4.3.1 Interpolation 1: Nearest Neighbor

Nearest neighbor is the simplest of all interpolation algorithms. It uses the pixel value from the source image that is the closest spatially to the target pixel's location. Following the graphic in Figure 4.7, we can visually determine that pixel (3,1) is the "closest" to the center of the target pixel. In this case, for nearest neighbor interpolation, the resulting value of the target pixel would simply be the exact value of pixel (3,1). This method is computationally efficient, but it has some severe drawbacks, especially when the sizes of the target and source image are very different.

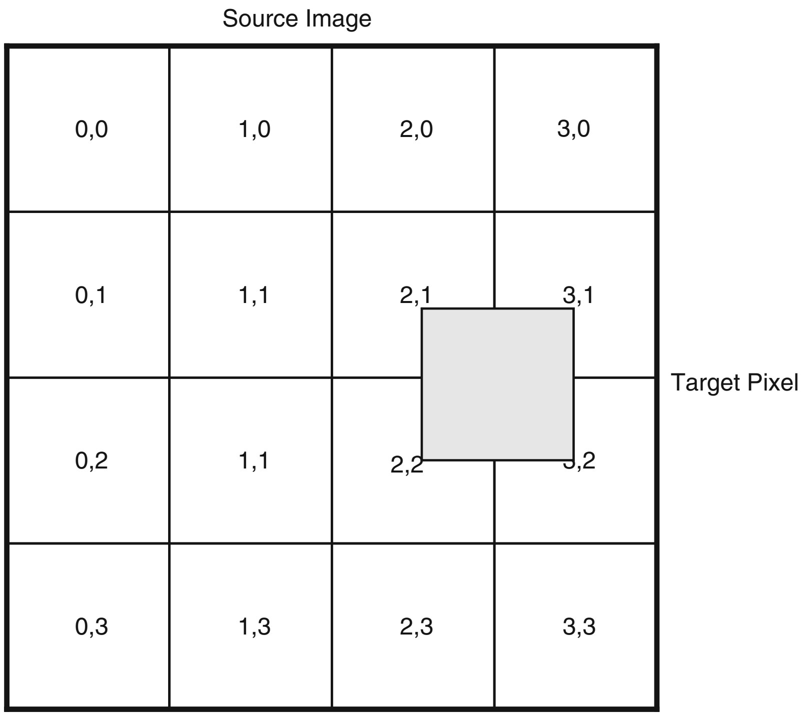


Fig. 4.7 Nearest neighbor interpolation uses only the closest pixel (3,1) to determine the target value.

Listing 4.6 shows the algorithm for nearest neighbor interpolation. The real work in this function is done by the "round" function, which simply rounds a real value to the closest integer.

Listing 4.6 Nearest neighbor interpolation.

```
def nearest_neighbor(s, sminx, sminy, smaxx, smaxy, source_width, source_height
    , tx, ty):
    i = round((tx - sminx) / (smaxx - sminx) * source_width)
    j = source_height - 1 - round((ty - sminy) / (smaxy - sminy) *
        source_height)
    return s[j][i]
```

Listing 4.7 Bilinear interpolation.

```
from math import *

def bilinear(s,sminx,sminy,smaxx,smaxy,source_width,source_height,tx,ty):
    temp_x = (tx - sminx) / (smaxx - sminx) * source_width
    temp_y = source_height - 1 - ((ty - sminy) / (smaxy - sminy) *
        source_height)

    i = floor(temp_x)
    j = floor(temp_y)
    weight_x = temp_x - i
    weight_y = temp_y - j
    val_00 = s[j][i]
    val_01 = s[j][i+1]
    val_10 = s[j+1][i]
    val_11 = s[j+1][i+1]

    pixel_val = (1 - weight_x) * (1 - weight_y) * val_00 + weight_x * (1 -
        weight_y) * val_01 + (1 - weight_x) * weight_y * val_10 + weight_x *
        weight_y * val_11

    return pixel_val
```

4.3.2 Interpolation 2: Bilinear

Bilinear interpolation is a little more complicated; it creates a weighted average of the 4 pixels which surround the center of the target pixel (Listing 4.7). Recall Figure 4.6; the bilinear interpolation would use pixels (2,1) , (3,1), (2,2), and (3,2).

Figure 4.8 illustrates the computations in the bilinear algorithm. The arrowed lines go from the center of the source pixels to the center of the target pixel. The length of each line, in ratio to the sum of the lengths, forms the complement of the weight given to the data from the pixel in which the line originates. It forms the complement because we want pixels with greater length to have less impact on the final result. They are ”further away” from the target pixel.

Let’s consider a variation on our bilinear algorithm. Suppose that our target pixel covers a large area in our source image, as in Figure 4.9. In this case, the bilinear algorithm would only use pixel data from pixels (1,1), (2,1), (1,2), and (2,2). Data from the other pixels would be disregarded. There are several solutions to this problem. The easiest is to perform multiple interpolation steps. Divide the target pixel into four (or more) sub-pixels and then perform a bilinear interpolation for each sub-pixel. When that is complete, compute the final target pixel value by bi-

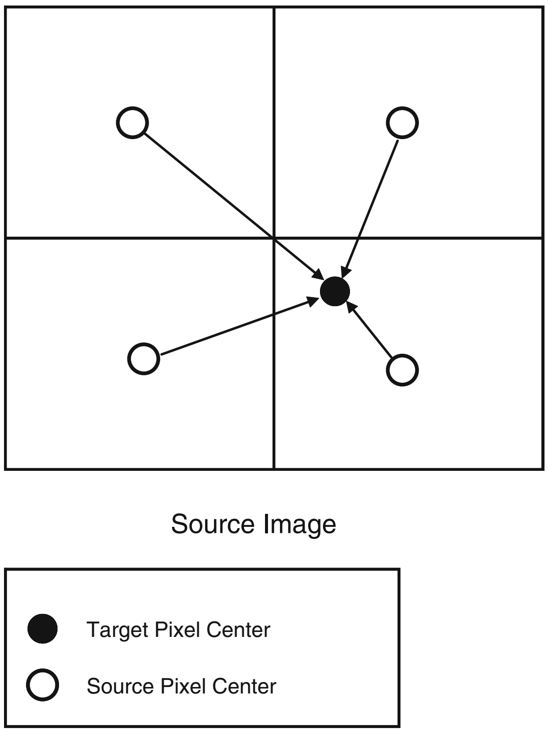


Fig. 4.8 Bilinear interpolation uses the four surrounding pixels to compute the target value.

linear interpolation over the four sub-pixels. This type of multi-step (also called multi-resolution) interpolation is the best way to handle image scaling that shrinks an image by a significant amount (Figure 4.10).

4.3.3 Interpolation 3: Bicubic

Bicubic interpolation is the most complicated of our interpolation algorithms. Where the bilinear interpolation considered the linear relationship of the 4 pixels surrounding our target point, the bicubic algorithm computes a weighted average of the 16 surrounding pixels. Figure 4.11 shows the target pixel with 16 surrounding pixels. Even though the outer 12 pixels do not overlap the target pixel, they are used for computing the surrounding gradients (or derivatives) of the pixels that do overlap the target pixel. This does not necessarily produce a more accurate interpolation, but it does guarantee smoothness in the output image.

The one-dimensional cubic equation is as follows:

$$f(x) = a_0x^3 + a_1x^2 + a_2x + a_3$$

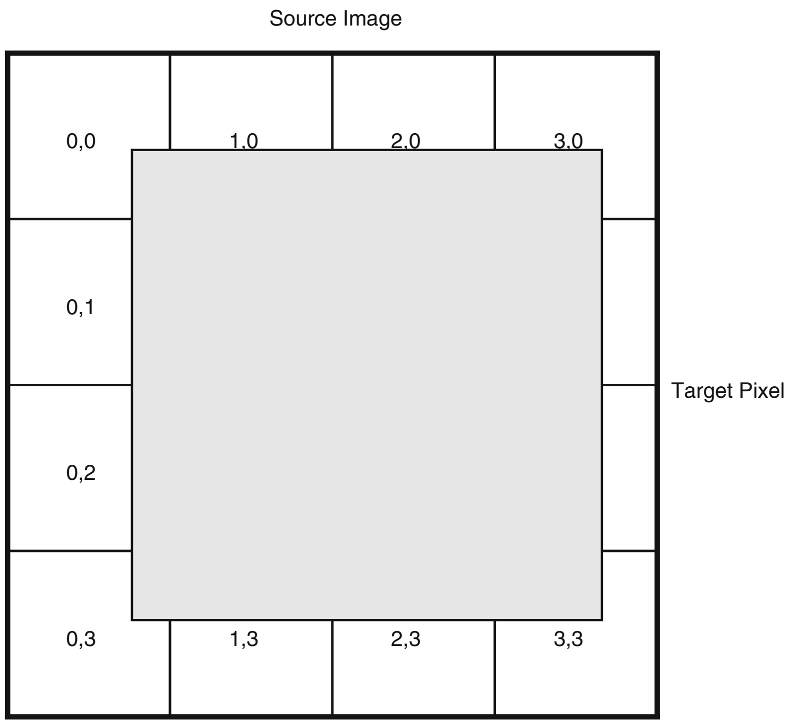


Fig. 4.9 Bilinear interpolation where the target pixel covers a large pixel area of the source image.

There are four coefficients: a_0, a_1, a_2, and a_3. The two-dimensional cubic equation, $f(x,y)$ has 16 coefficients, a_{00} through a_{33}. There are several ways to compute the 16 coefficients using the 16 pixel values surrounding the target pixel. Most involve approximating the derivatives and partial derivatives to develop a set of linear equations and then solving the linear equations. The full explanation of that process is beyond the scope of this chapter. We suggest the references *Numerical Recipes in C* and ”Cubic convolution interpolation for digital image processing” for more information [3, 2].

Since each interpolation algorithm has different performance characteristics, we will examine the results with real images. Figure 4.12 is an image of a fish.[1] If we scale a small section of the fish's scales to 400% (or 4 times magnification) in each dimension, we get the images shown in Figure 4.13, Figure 4.14, and Figure 4.15 for nearsest neighbor, bilinear, and bicubic interpolations, respectively. In this example, only the bicubic interpolation yields a satisfactory result.

We will also consider an example using a rendered map graphic. Figure 4.16 is a map of a portion of the city of New Orleans from OpenStreetMap.[2] We will use

[1] Fish images courtesy of Robert Owens, Slidell, Louisiana.

[2] OpenStreetMap images used from www.openstreetmap.org.

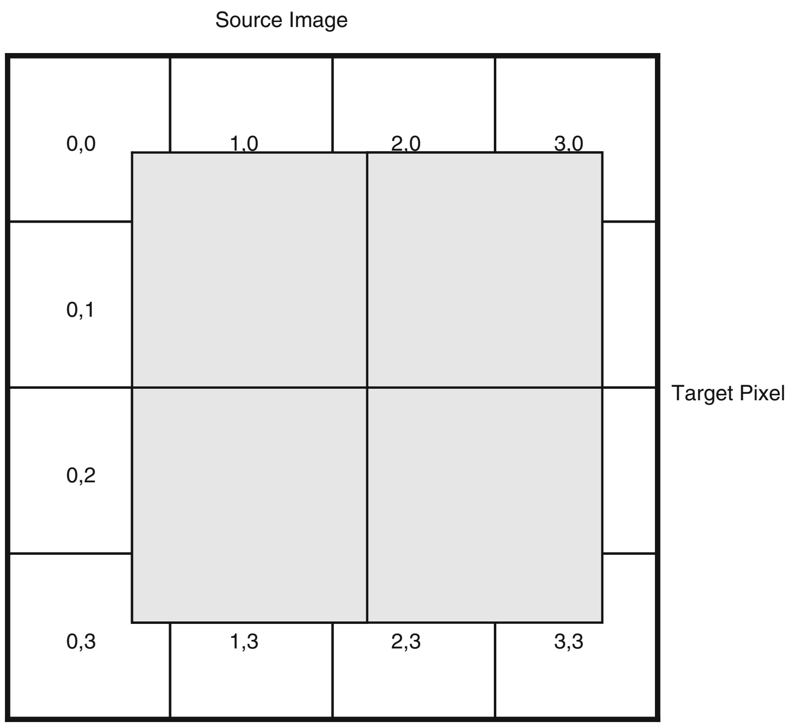


Fig. 4.10 Bilinear interpolation can be performed in multiple steps to compute target pixels that cover many source pixels.

our interpolation algorithms to scale a sub-section of that image. We have chosen a subsection with lots of lines and text. These are typical features in map images. If we scale a small section of the image to 400% (or 4 times magnification) in each dimension, we get the images shown in Figure 4.17, Figure 4.18, and Figure 4.19 for nearsest neighbor, bilinear, and bicubic interpolations respectively. Once again, only the bicubic provides a satisfactory result. Figure 4.20 shows a section of the image with text highly magnified by bicubic interpolation. Figure 4.21 shows a section of the image with text highly magnified by bilinear interpolation. The bicubic interpolation performs much better with text features.

Text features are especially sensitive to interpolation. Even though the bicubic interpolation imposes a significant performance penalty, it is probably worth the cost in most cases.

Listing 4.9 shows implementations of the nearest neighbor and bilinear interpolation algorithms for RGB images. The classes BoundingBox and Point2DDouble are simply wrapper classes for multiple coordinates. BufferedImage is the Java built-in class for manipulating image data. Many programming environments provide built-in tools for scaling and subsetting images. This changes our algorithms slightly. Instead of performing pixel-by-pixel calculations, we compute a single set of trans-

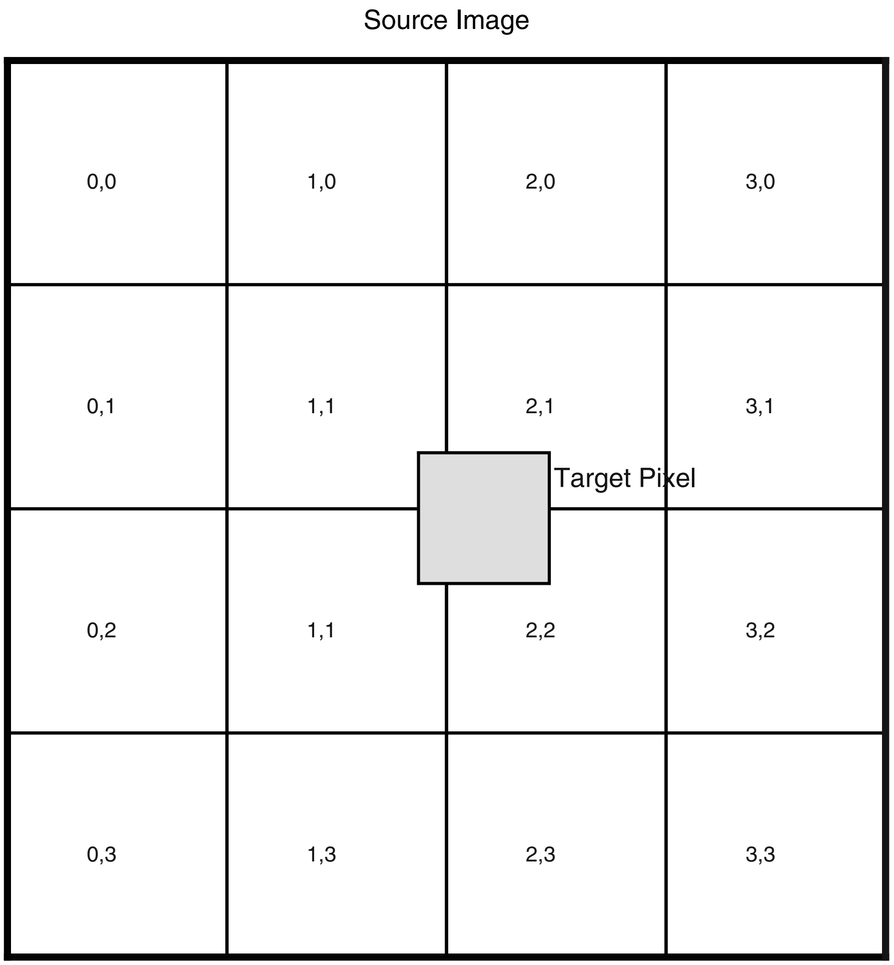


Fig. 4.11 Bicubic interpolation uses the 16 surrounding pixels to compute the target pixel value.

formation parameters and pass those to the built-in image manipulation routines. Listings 4.10 and 4.11 show how to use those built-in routines in Java and Python.

Practical experience has shown that bilinear interpolation takes approximately 150% the time as nearest neighbor, and bicubic interpolation takes approximately 200% the time as nearest neighbor.

The astute reader will notice that we have used bilinear interpolation throughout our discussion as the means of calculating the geographical coordinates. The supplied algorithm ”geolocate” uses bilinear interpolation to map between geographic and pixel coordinates. So why is it good enough to use bilinear for calculating geographic coordinates but not good enough for calculating the actual pixel values? The mapping from geographical coordinates to pixel space and back is, by definition, a

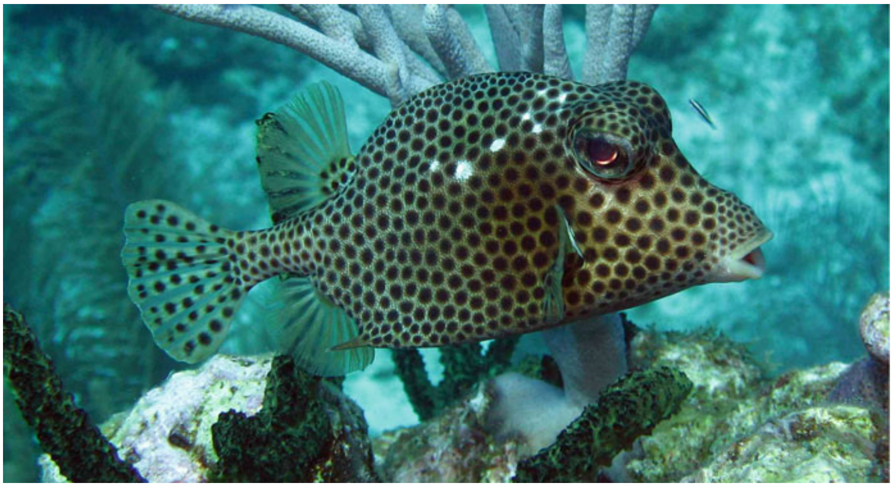


Fig. 4.12 Fish Image

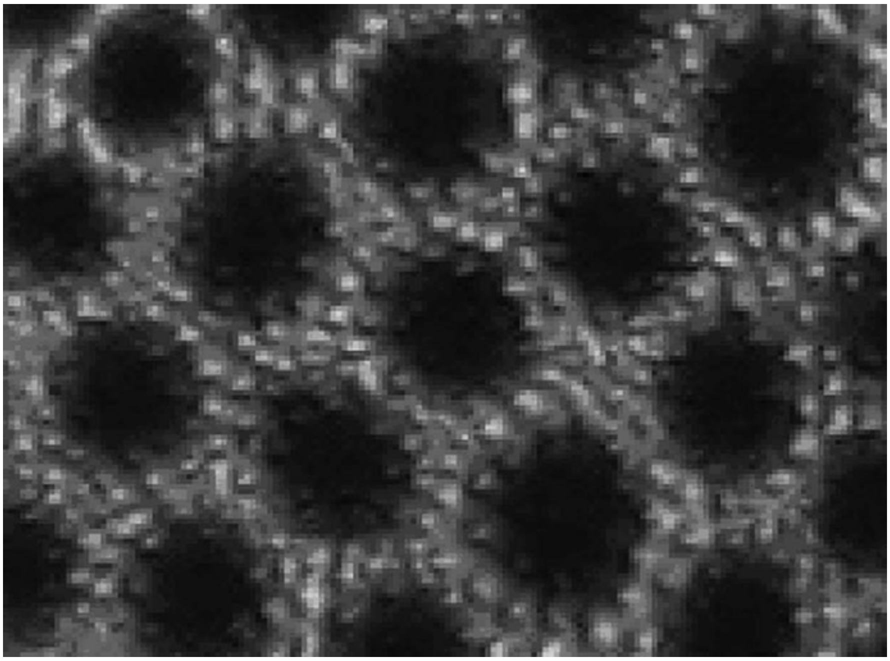


Fig. 4.13 Fish Scale image magnified with nearest neighbor interpolation.

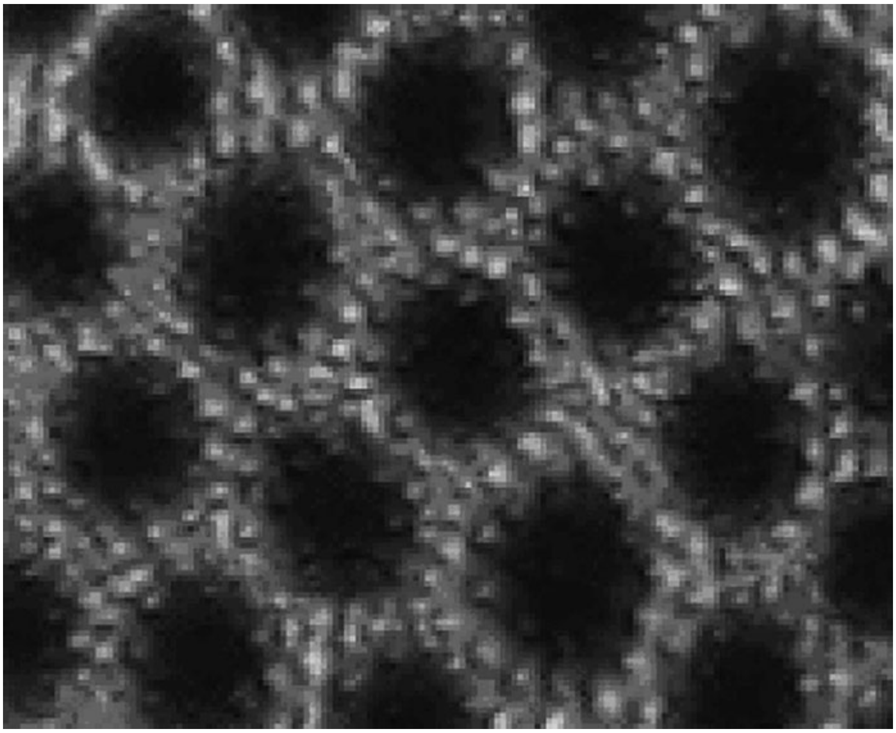


Fig. 4.14 Fish Scale Image magnified with bilinear interpolation.

linear function. We can compute the exact transformation from geographic coordinates to pixel coordinates. However, the contents of the image, the actual color values of pixels, are highly non-linear. Whether the image contains aerial imagery or a rendered map graphic, there is very little linearity, either locally or globally, between the actual values of the pixels.

4.4 Choosing Image Formats for Tiles

Any tile-based mapping system must use image file formats for storage and transmission of image tiles. There are hundreds of file formats that can be used. Some offer very sophisticated compression schemes, and others focus on simplicity and compatibility. We want to choose image formats that can be encoded and decoded quickly, offer good compression performance, and, most importantly, are supported natively by common web browsers.

It is possible that we will use one format to store images and another to transmit them. In general, we want to reduce image processing and manipulation tasks that

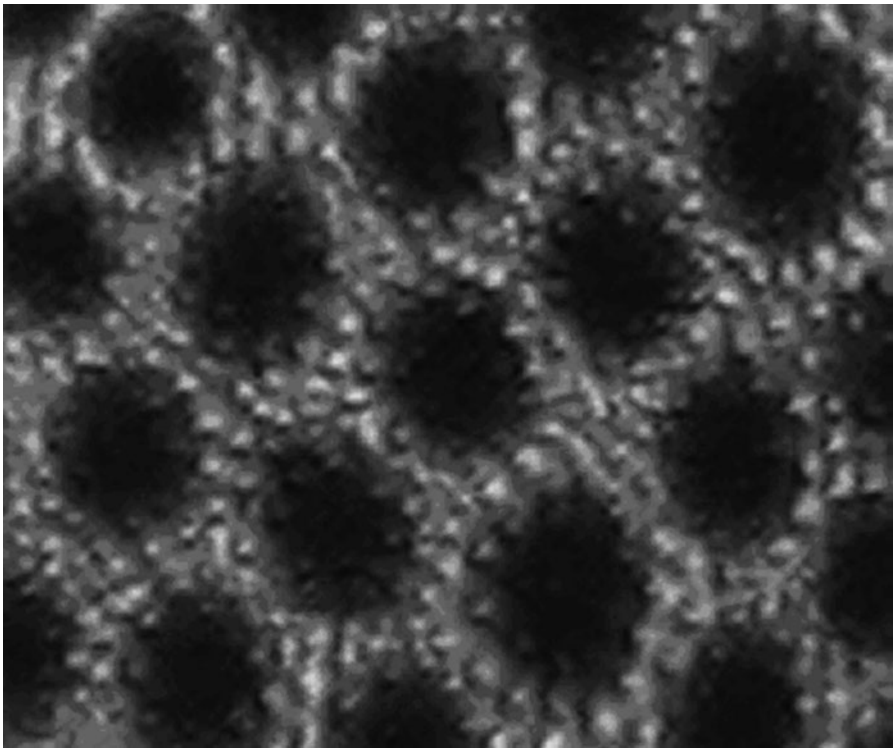


Fig. 4.15 Fish Scale Image magnified with bicubic interpolation.

are required for each client access. Our goal is to use the same format for storage and retrieval.

Table 4.2 lists several popular browsers and their supported image formats. Native browser support is critical. Browser-based (HTML/JavaScript) map clients, like OpenLayers and Google Maps, achieve their quick performance and appealing look by using the native capabilities of the browser to display and manipulate images. If we adopt formats that are not well supported by the majority of Web browsers, we have needlessly crippled our system's performance. From the table, we can see that JPEG, GIF, BMP, and PNG are commonly supported. Table 4.3 shows the features of each format.

Browser	JPEG	JPEG2000	GIF	TIFF	BMP	PNG
Internet Explorer	Yes	No	Yes	No	Yes	Yes
Firefox	Yes	No	Yes	No	Yes	Yes
Google Chrome	Yes	No	Yes	No	Yes	Yes
Safari	Yes	Yes	Yes	Yes	Yes	Yes
Opera	Yes	No	Yes	No	Yes	Yes

Table 4.2 Browser support for different image compression types.

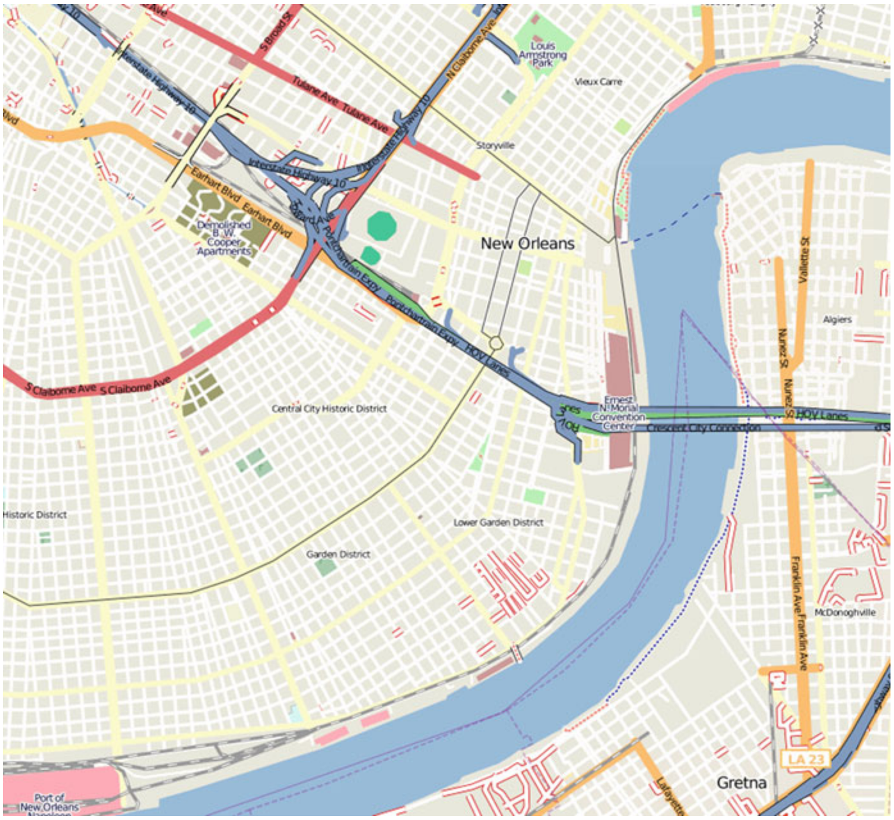


Fig. 4.16 Rendered map of New Orleans.

Format	Compression	Colors Supported	Transparency Supported
JPEG	Lossy	24 bit RGB	No
GIF	Lossless	8 bit Indexed	Yes
BMP	Uncompressed	24 bit RGB	Yes
PNG	Lossless	48 bit RGB	Yes

Table 4.3 Details of different compression types.

We can eliminate BMP from consideration since it is not compressed. Also, we can eliminate GIF because it does not lend itself to full 24 bit color. This leaves us with PNG and JPEG. PNG provides lossless compression and support for transparency while JPEG provides lossy compression.

PNG uses the DEFLATE lossless compression algorithm. PNG can achieve superior compression with images that have few unique colors, repeated pixel patterns, and long sequences of the same pixel value. As such, it is quite suitable for storing rendered figures and maps that typically have limited color palettes.

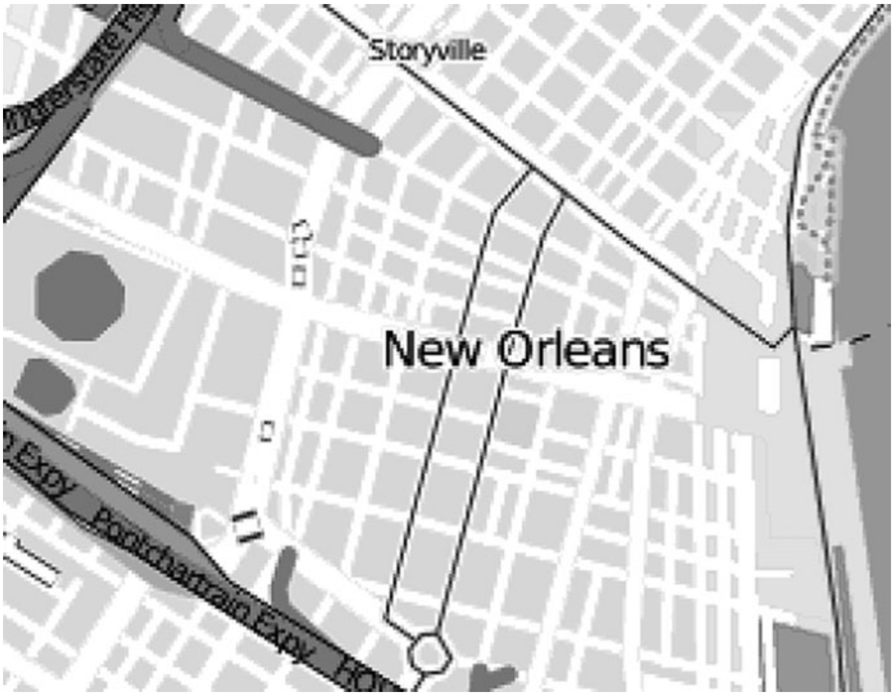


Fig. 4.17 New Orleans map subsection with nearest neighbor interpolation.

JPEG uses a Discrete Cosine Transform based compression algorithm. It performs well with images that have lots of colors, some noise, and softer transitions typically found in photography.

As the basis for a comparison of PNG and JPEG performance characteristic, recall our fish image, Figure 4.12. The color version of this image has 315,559 colors and 2088 by 1128 pixels. Stored as a JPEG, the file is 147kb. Stored as a PNG, the file is 2.67mb. That is a ratio of around 18 to 1. That means if we use PNG storage for our tiles we will need 18 times the storage space, and our users will have to wait 18 times longer for the images to download. The color version of our rendered map graphic of New Orleans, Figure 4.16, has 2372 colors and 780 by 714 pixels. The PNG version is 321kb, and the JPEG version is 113kb. This is a much more reasonable 3 to 1 ratio.

While not visible at the default scales, compression artifacts are visible where there are sharp color boundaries in the image. Figure 4.22 is a test image with some text saved as a JPEG with the default quality settings. Compression artifacts are visible at the text boundaries.

Based on these considerations, we provide the following guidance.

- Use JPEG images when dealing with aerial or satellite photography, images with lots of colors, or when storage space is a critical issue.

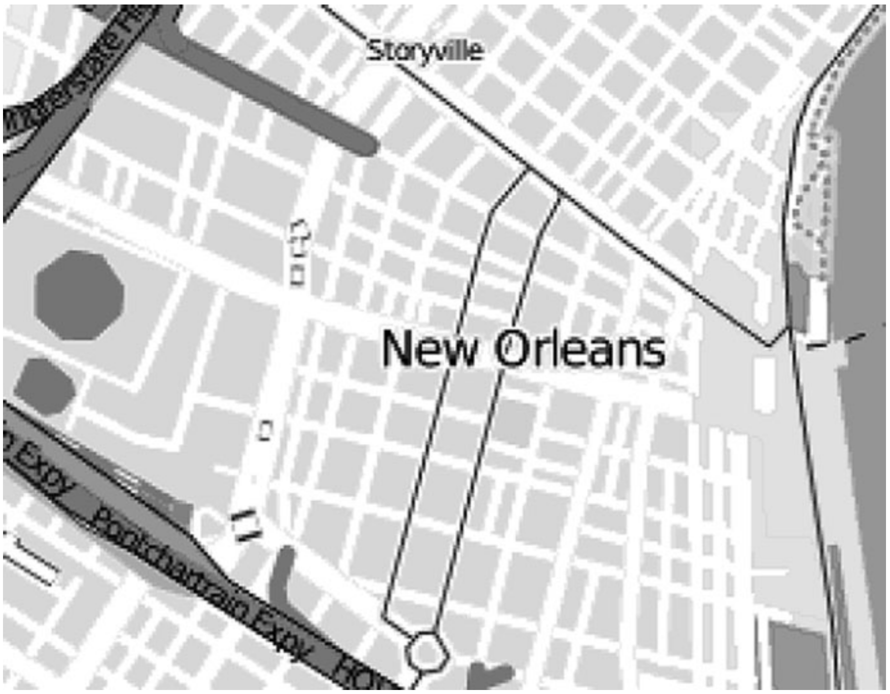


Fig. 4.18 New Orleans map subsection with bilinear interpolation.

- Use PNG images when transparency is required or when quality of reproduction for rendered map graphics is critical.

Both PNG and JPEG include, within the first few bytes of the file format, a unique identifier that allows image reading software to know the format of the file. This self identification property simplifies tile storage, since the tile storage system does not need to store the file type that was used to store a tile.

It is perfectly reasonable to use both formats together in the same tile system. For example, a tiled map layer that has data for only a small portion of the earth would use a transparency enabled format for the low-resolution scales so that map users could see the covered areas in the conjunction with other background layers. It would then switch to JPEG for the high-resolution images that have larger storage requirements.

The reader may ask why we have not chosen to use one the common geospatial image formats for storing our tiles. The answer is simple. First, tiles have their geospatial coordinates embedded in their tile address. Recall from the discussion in Chapter 2 on logical tile schemes that a tile scheme provides for conversion from a tile's address to its map coordinates and back. Secondly, and most importantly, geospatial image formats are not commonly supported by Web browsers.

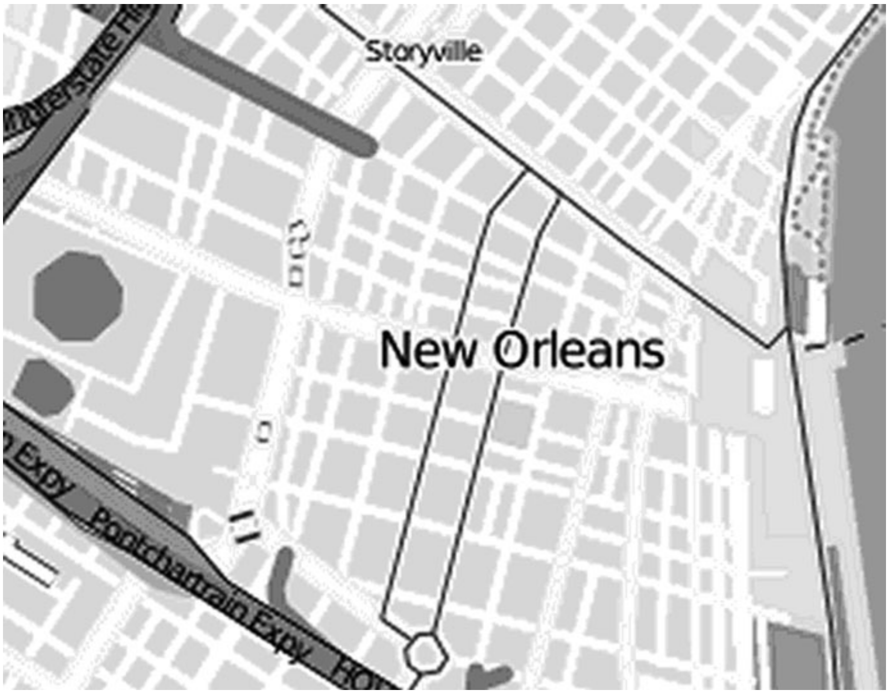


Fig. 4.19 New Orleans map subsection with bicubic interpolation.

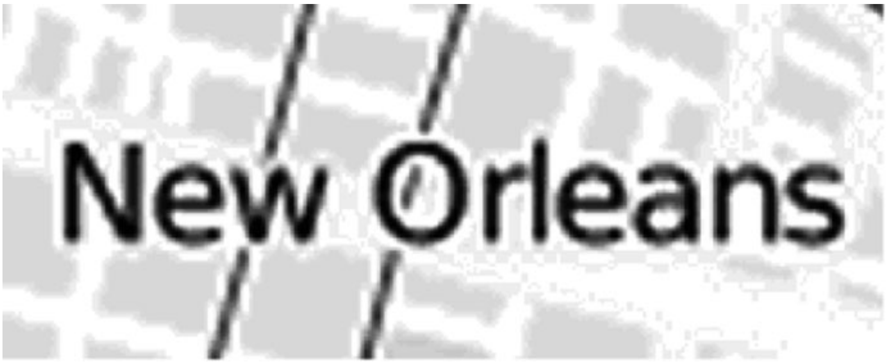


Fig. 4.20 Close up text with bicubic interpolation.

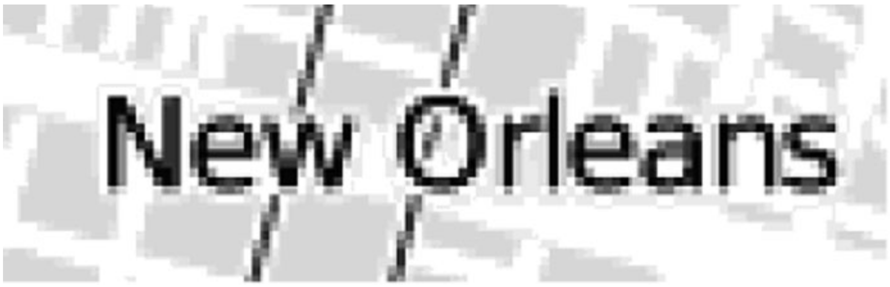


Fig. 4.21 Close up text with bilinear interpolation.

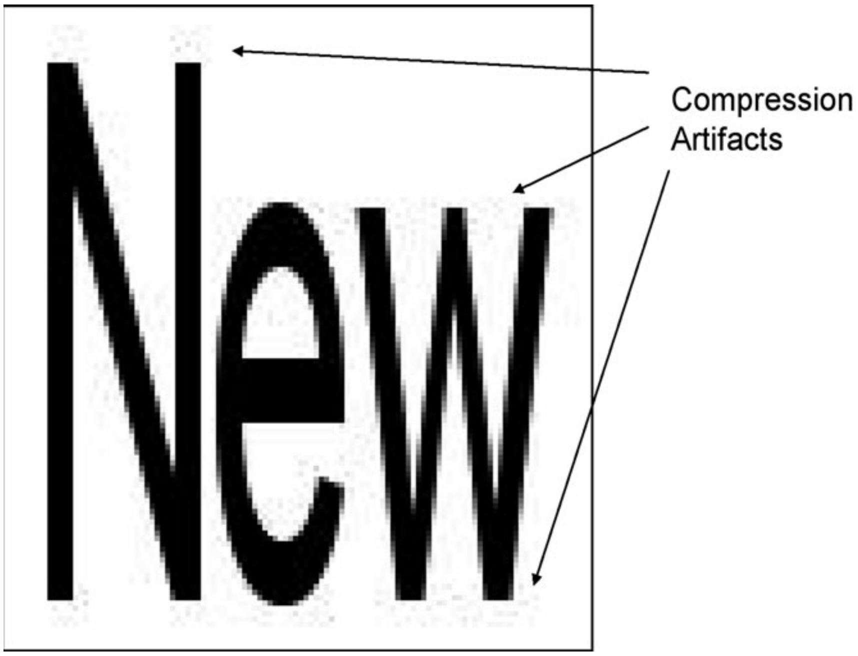


Fig. 4.22 JPEG compressed text with artifacts.

4.5 Choosing Tile Sizes

The choice of tile image dimensions is one of the most important decisions to be made in the design of a tile-based mapping system. Tile images can be any size, and they can vary from scale to scale. They can also vary across the same scale, or they can be random in size. However, there are efficiencies with making tiles uniform in size across each and every scale. Also, there are efficiencies from choosing tiles that have the same horizontal and vertical dimensions. Furthermore, tile sizes that are powers of two yield simpler mathematics throughout the process.

There are several approaches to determining the optimal tile size. First we should consider the impact of using multiple images to virtualize a single map view. Each image comes with a certain amount of overhead. There are several types of overhead involved that include the overhead of multiple seeks and reads from the computer's file system, uneven utilization of the file system's native block size, and the header and other overhead storage space within each image file.

Let us consider the constraints of current image formats. We have limited ourselves to image formats that are readily usable by most Web browsers: JPEG and PNG. Any encoded image is going to use space for overhead, i.e. space not directly used to store pixels. This is header information and image metadata. Some example images will allow us to inspect the overhead of the JPEG and PNG formats. We generate images with scaled content of sizes 1 by 1, 64 by 64, 128 by 128, 256 by 256, 512 by 512, 10214 by 1024, 2048 by 2048, 4096 by 4096, and 8192 by 8192 pixels. We are using a segment of NASA's Blue Marble Next Generation as our source content and our 1x1 pixel image as the baseline.

Image Size	JPEG Bytes	PNG Bytes	JPEG Overhead Percentage	PNG Overhead Percentage
1 x 1	632	69	100.0%	100.0%
64 x 64	2019	8532	31.30%	0.81%
128 x 128	4912	30724	12.87%	0.22%
256 x 256	14267	111642	4.43%	0.06%
512 x 512	43424	410782	1.46%	0.017%
1024 x 1024	135570	1515218	0.47%	0.0046%
2048 x 2048	423298	5528685	0.15%	0.0012%
4096 x 4096	1309545	19513354	0.048%	0.00035%
8192 x 8192	4549578	62798290	0.014%	0.00011%

Table 4.4 Comparison of JPEG vs PNG compression performance.

Clearly, we can reduce overhead by using very large images. But very large images introduce a new problem. It is unlikely that our users will be very satisfied waiting for a 8192 by 8192 image to download and display, especially when their monitors can show only 1024 by 768. They are able to view only 1.17% of the pixels in the image at one time. Also, very large images consume a lot of system memory and may not be usable at all on smaller or older devices.

There is another consideration to be made that is specific to JPEG images. The JPEG compression algorithm is block based. It commonly uses 16 by 16 blocks of pixels as minimum compression units. If an image's pixels are not evenly divisible by 16 in each dimension, it will pad the image with empty values. We can prove this by creating a series of JPEG images sized 1 to 500 pixels. Each image consists of all black pixels.

The distinct stair-step pattern in Figure 4.23 shows that the images increase in compressed size by 16 pixel increments. Therefore, we should choose tile sizes that are powers of 16, like 16, 32, 64, etc. This partially explains why our overhead calculations for JPEG and PNG images showed that overhead as a percentage from

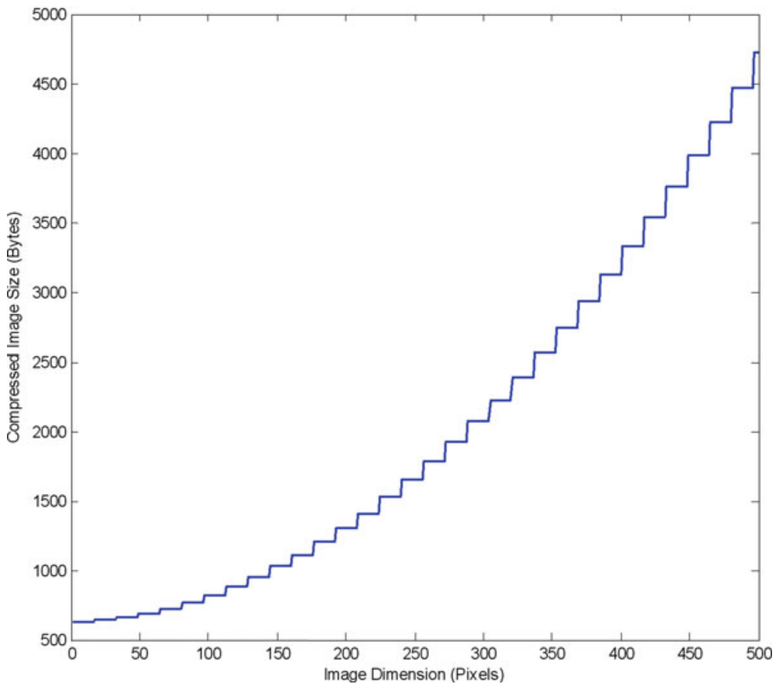


Fig. 4.23 Graph shows the step-pattern for size of a JPEG-compressed image.

JPEG images is much larger than PNG images. The 1 by 1 JPEG image would be the same size as a 16 by 16 JPEG image. This would not be the case for PNG images.

To determine the actual appropriate tile size, we can create an optimization function. We want to minimize both the number of individual images required to virtualize the map view and the number of wasted pixels. Wasted pixels are pixels that are transmitted and decoded but not part of the virtualized map view (See Figure 4.24).

The best way to minimize wasted pixels would be to make all of our tiles 1 by 1, and then we would never have to decode any pixels that are not part of the final image. However, the overhead of having to retrieve and decode thousands and thousands of image files per map view would make our system unusable.

We can experimentally determine the proper tile size for our system. First we need to guess the typical size of a virtualized map view. For this example, we will use 1024 by 768 pixels. Given this size, we can generate a large number of random map views for a given scale. For each of those random map views, we will calculate the number of tiles needed to fill that that view and the number of wasted pixels. We will perform this calculation for all the tile dimensions that we are considerating using. For this example, we will use tile dimensions 16, 32, 64, 128, 256, 512, 1024, and 2048.

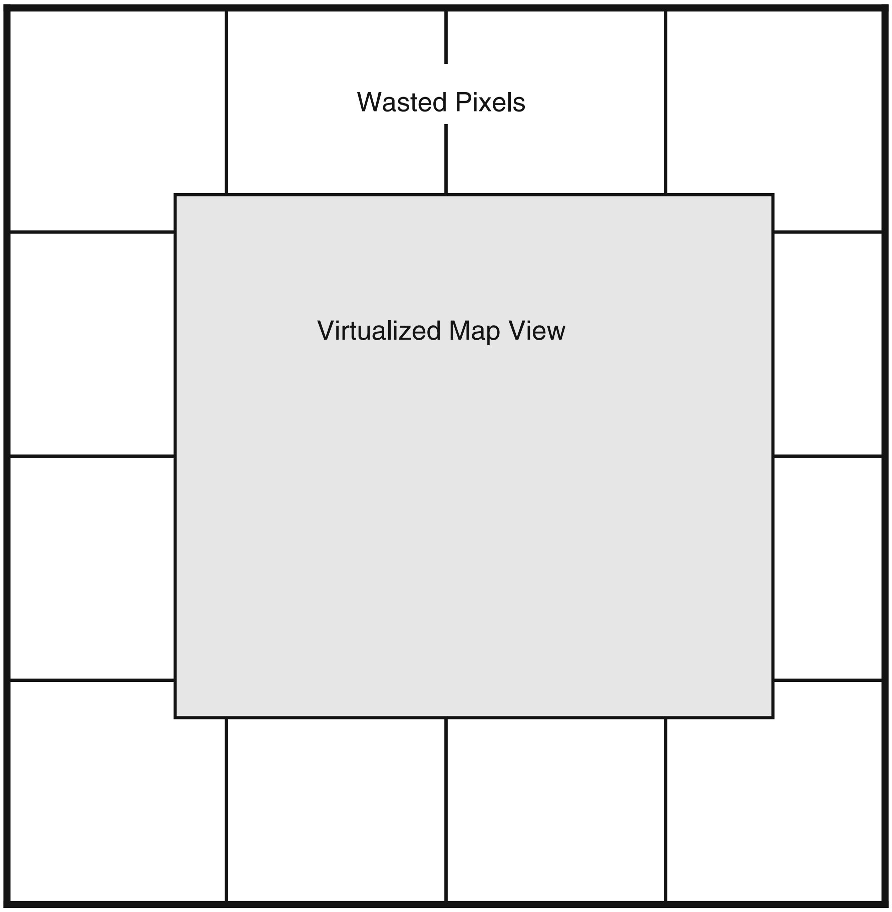


Fig. 4.24 Wasted pixels are decoded but not used as part of the virtualized map view.

For an example map scale, we will use scale 10 from the logical tile scheme that we developed in Chapter 2. Recall, that scale 10 is simply defined as having 2^{10} (1024) columns by 2^9 (512) rows. Each tile is $\frac{1024}{360.0}$ ($2.8\overline{4}$) degrees wide and long.

Generating random map views is fairly easy. Since all map views have the same aspect ratio, we need to generate a large number of random center locations. The center locations would be in the range of -180 to 180 for the longitude coordinate and -90 to 90 for the latitude coordinate. Then, for each tile size, we can extrapolate the map view bounds from the center location.

When generating our random map views, we have to consider that cases in which some portion of our map view will go beyond the normal bounds of the Earth. For example, the longitudes might be greater than 180.0 or less than -180.0. There are two ways to deal with this. First we can constrain our randomization function to a range of coordinates that is guaranteed to never generate map view bounds outside our given range, or, secondly, we could simply perform those calculations without caring if the boxes overlap our acceptable coordinates. We will choose the latter method. If a randomly computed map box strays beyond the -180 to 180 and -90 to 90 bounds, we will compute wasted pixels and tiles that were accessed as if there were tiles and pixels in those areas. This is a practical decision because many map clients perform wrapping in boundary areas. They pull images and pixels from the other side of the map to fill in the boundary overlaps.

The algorithm in Listing 4.12 generates 10,000 randomized map center locations for scale 10 and tile dimensions of 16, 32, 64, 128, 256, 512, 1024, and 2048. It computes the total number of tiles accessed and the wasted pixels for each access. Those results are shown in Table 4.5.

Tile Size	Number of Tiles Accessed	Wasted Pixels
16	3185.0	28928.0
32	825.0	58368.0
64	221.0	118784.0
128	63.0	245760.0
256	20.0	524288.0
512	7.5021	1180198.5024
1024	3.4938	2877082.8288
2048	2.0677	7886130.3808

Table 4.5 Tiles accessed and wasted pixels for 1024 by 768 map view. 10,000 random map views averaged.

When the results are plotted, it is easy to see the optimal point, as shown in Figure 4.25. We have normalized the tiles accessed and pixels wasted values. The two lines cross very near to when the tiles are sized 128 by 128. This statistic might lead us to select tiles sized 128 by 128. However, these calculations are performed in pixel counts. We are disregarding the important computations performed earlier to determine overhead percentages for each tile size. Re-computing the optimization and substituting pixels wasted with total bytes accessed yields a different result. Furthermore, the result can be plotted with just one line to see the bytes used as a function of tile size; see Figure 4.26. Table 4.6 shows our results using the listed tile image sizes in bytes.

Tile Size	Standardized Image Size in Bytes	Number of Tiles Accessed	Total Bytes Accessed
16	759	3185.0	2417415.0
32	1062	825.0	876150.0
64	2019	221.0	446199.0
128	4912	63.0	309456.0
256	14267	20.0	285340.0
512	43424	7.5	326070.816
1024	135570	3.5	474305.202
2048	423298	2.06	873687.072

Table 4.6 Bytes accessed for different sized tiles.

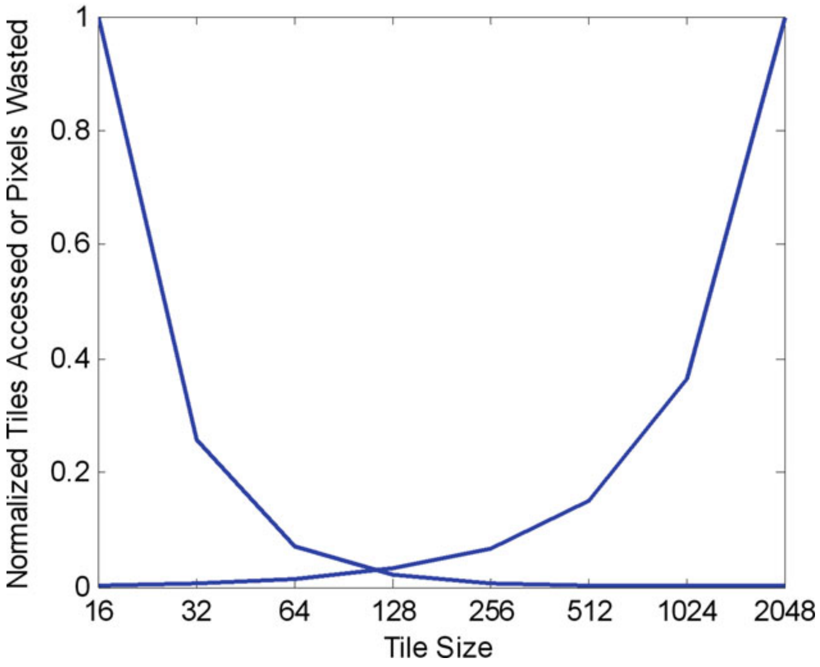


Fig. 4.25 Graph of the normalized number of tiles accessed and of pixels wasted.

Clearly the 16 by 16 tiles are very inefficient. They require the most bytes to be read, even though our earlier computations showed that they generated the fewest wasted pixels. The effect of the wasted pixels is seen as the tile sizes get larger. According to this graphic, tiles sizes 128, 256 or 512 are all close to optimal.

What if we consider more than one map view resolutions? Up to now, we have considered only 1024 by 768 map view resolutions. Figure 4.27 shows the results for map resolutions 640 by 480, 800 by 600, 1024 by 768, 1280 by 960, 1400 by 1050, and 1600 by 1200.

The results are similar: we still see the bottom (or optimum area) of our plots around the 128, 256, and 512 area. Figure 4.28 shows the results for PNG image's sizes instead of JPEG sizes. We can see the effect of reduced overhead in PNG images, but otherwise the plots are similar.

Figure 4.29 shows the JPEG bytes accessed plotted as differences from one tile size to the other. In this figure we can see that the line is almost flat from 256 to 512. This indicates that there is very little difference between these two tile sizes in terms of total bytes accessed.

Since we have moved from considering pixels to compressed image bytes, we should also consider computation time required to decompress the compressed tile images. Table 4.7 shows the average decode times for tiles of varied sizes in both

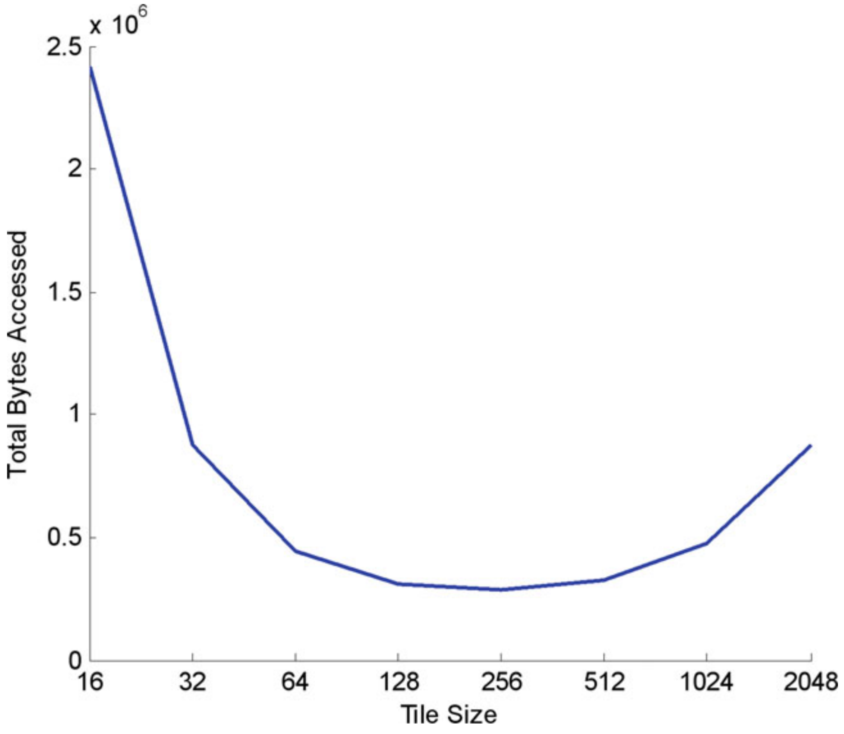


Fig. 4.26 Graph of bytes accessed vs. tile size.

JPEG and PNG formats. If we put these numbers into our previous optimization plots we get Figure 4.30 and Figure 4.31. In both of these experiments, we see that our plot has a minimum around the 512 tile size.

Tile Size	Decode Time JPEG (Milliseconds)	Decode Time PNG (Milliseconds)
16	2.5	2.34
32	2.5	2.5
64	2.66	3.12
128	3.91	4.22
256	5.0	7.97
512	10.94	21.1
1024	31.56	69.22
2048	113.75	258.44

Table 4.7 Decode times for JPEG and PNG tiles.

Further enhancements to this approach are possible. We have considered only random map views that exactly match our pre-determined map scales. In practice this will occur only for map clients that adhere to those fixed map scales. In addition, we have fixed the map scale by number of tiles and varied the tile size. In practical

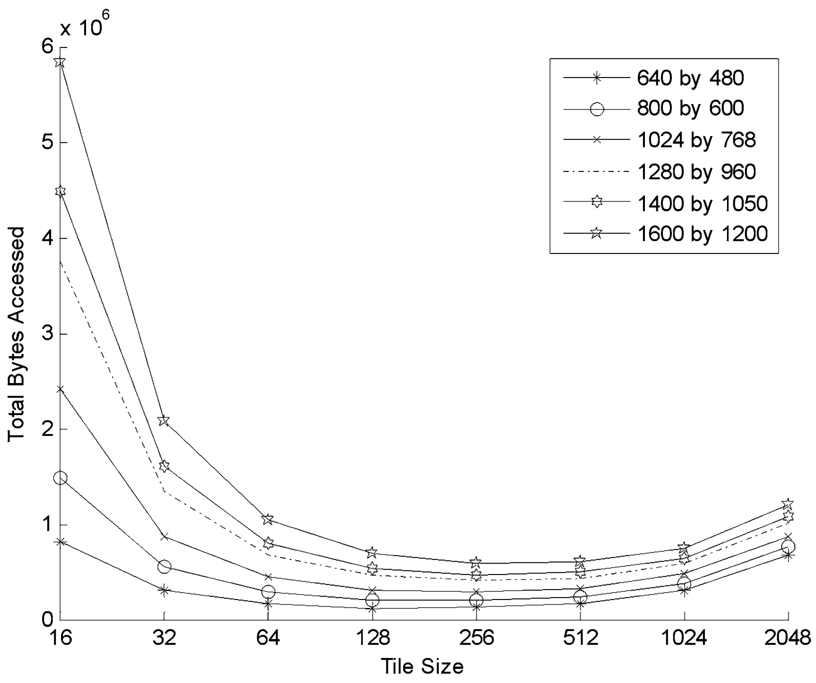


Fig. 4.27 Graph of bytes accessed using JPEG tiles for multiple map display resolutions.

terms this means that while each tile covers that same portion of the earth, the real map resolution of each tile varies with its size. Both of these shortcomings can be addressed by replacing our fixed map scale of 10 with a randomly selected map scale. The randomly selected map scale should be chosen from a continuous range instead of fixed discrete scales. In these cases, we will have to scale the pixels from the covered tiled region to match the scale of the map view with the randomly chosen scale.

In conclusion, from consideration of the results given above we are going to use 512×512 for the tile sizes in this book. Our analyses indicate the 256×256 would also be a good choice. However, we should consider a final point. It takes four 256×256 tiles to cover the area of one 512×512 tile. Thus when we create a large number of tiles, if we use 256×256 tiles, we will have four times the database entries or four times the tile image files, and either way our indexes will be four times the size. As we cover techniques for producing and storing tiles, we will see that these are significant costs.

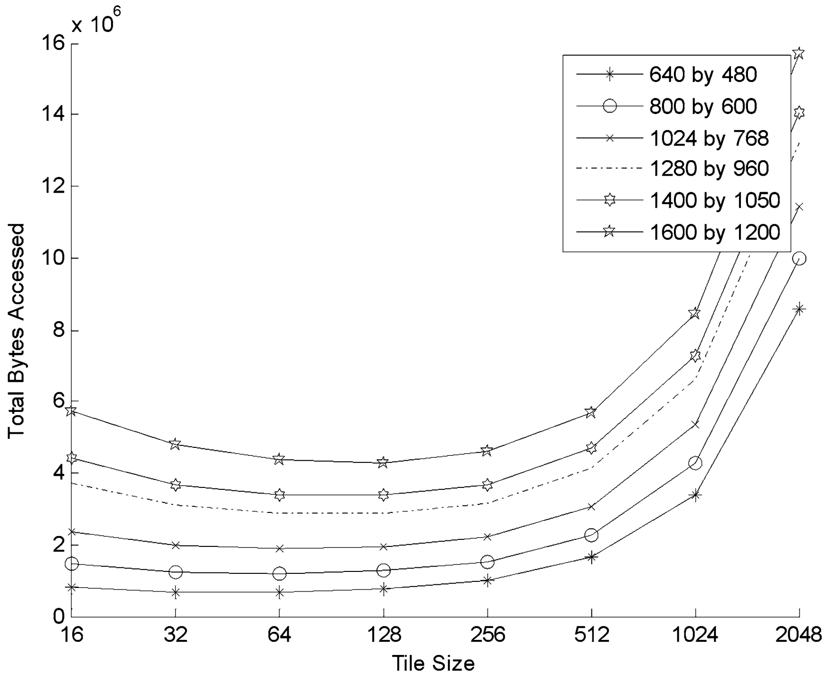


Fig. 4.28 Graph of bytes accessed using PNG tiles for multiple map display resolutions

4.6 Tuning Image Compression

In dealing with very large tiled image sets, it may be important to try to reduce the amount of storage space required. The compression quality of JPEG compressed images can be adjusted to produce smaller or larger compressed files. As the compression rate is increased the file size decreases. Many software platforms support setting the JPEG quality ratio to values in a pre-defined range; these can be either 0 to 1, or 0 to 100. Higher quality values mean less compression. If we apply varied quality settings to a 512 by 512 JPEG image taken from the Blue Marble data set, we get the file size differences shown in Table 4.8.

Quality Setting	JPEG File Size in Bytes
90	62128
80	45475
70	42141
60	38746
40	21967
10	9952

Table 4.8 File sizes for different JPEG quality settings.

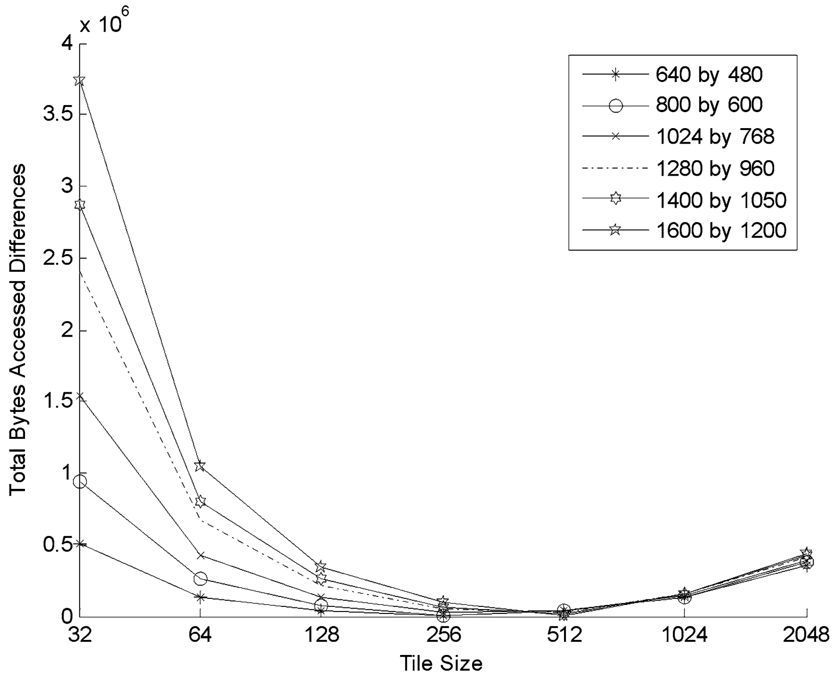


Fig. 4.29 Change in the number of bytes accessed as tile size increases (JPEG tiles).

Figure 4.32 is a tiled image from the Blue Marble set. Figures 4.33(a), 4.33(b), 4.33(c), 4.33(d), 4.33(e), and 4.33(f) show the results of applying the various compression quality settings.

The 80 and 90 quality settings are visually hard to distinguish, but lower values show compression artifacts. For example, the 10 quality image is quite blurry. There is a significant drop in storage space required from quality setting 90 to 80. After that, the drops are less pronounced. Anyone producing large tile sets should take time to manually set the quality setting appropriate to their application. Tile producers might consider using a lower quality setting for lower resolution tiles and using a higher setting for higher resolution tiles. This would provide tile system users with lower quality overview images, but the option to zoom in for higher quality map views.

Even though the PNG format is technically lossless, we can apply some lossy techniques to reduce the file sizes of PNG images. Recall that the PNG format is sensitive to the number of colors used in an image. If we can reduce the number of colors in an image, a process called "color quantization," we can reduce the size of the compressed PNG file.

There are many algorithms for color quantization. A very simple algorithm would simply reduce the byte space available for colors. So instead of an RGB image with 8 bits for each color channel, we could only allow 7 bits for each channel. Listing 4.8 shows this algorithm applied to an example map image.

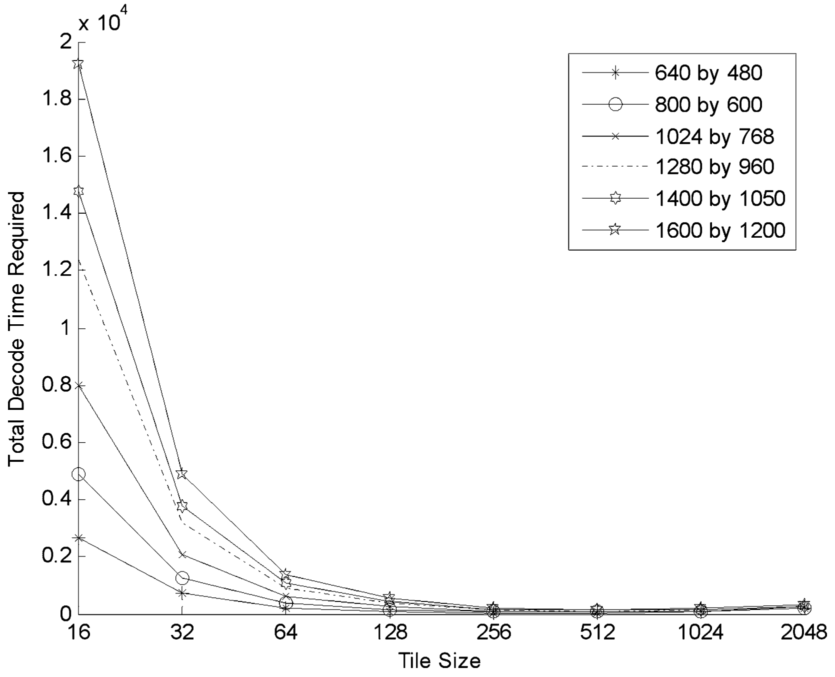


Fig. 4.30 Decode time for JPEG images as tile size increases.

Listing 4.8 Simple algorithm for reducing the color space of an image.

```

byte[] squeezeColors(BufferedImage bi, int bitsPerColor) throws IOException {
        int bitsToShift = 8 - bitsPerColor;
        for (int i = 0; i < bi.getWidth(); i++) {
            for (int j = 0; j < bi.getHeight(); j++) {
                Color c = new Color(bi.getRGB(i, j));
                int b = c.getBlue();
               int g = c.getGreen();
                int r = c.getRed();
                b = (b >> bitsToShift) << bitsToShift;
                g = (g >> bitsToShift) << bitsToShift;
                r = (r >> bitsToShift) << bitsToShift;
                Color c2 = new Color(r, g, b);
                bi.setRGB(i, j, c2.getRGB());
            }
        }
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        ImageIO.write(bi, "png", baos);
        byte[] data = baos.toByteArray();
        return data;
    }
```

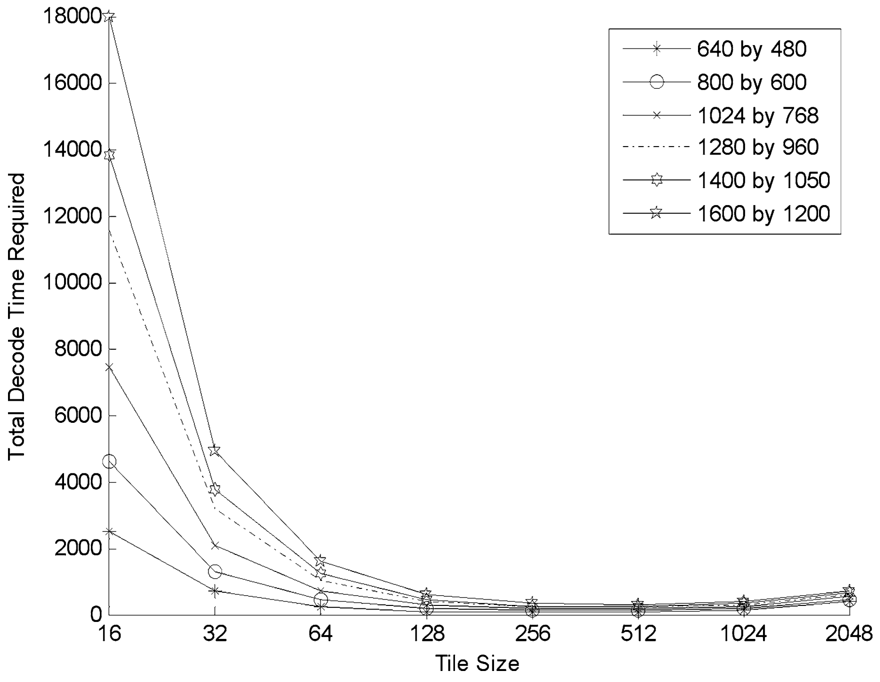


Fig. 4.31 Decode time for PNG images as tile size increases.

```
b = (b >> bitsToShift) << bitsToShift;
g = (g >> bitsToShift) << bitsToShift;
r = (r >> bitsToShift) << bitsToShift;
```

The key part of the code is the following section:

We shift each color component to the right and then back to the left. This has the effect of rounding off or zeroing out the right most or least significant bits for each component.

Table 4.9 shows the effect of this simple algorithm on the number of colors and the resulting file sizes on the image shown in Figure 4.16.

Reducing the color palette from 24 bits to 21 bits reduces the number of colors from 2,372 to 2,094, and it delivers a significant reduction in file size, nearly 45%. Figures 4.34(a), 4.34(b), 4.34(c), 4.34(d), 4.35(a), 4.35(b), and 4.35(c) show the effect of reducing the color palette. Only the most severe reductions provide a visible decrease in the image's quality.

There are more sophisticated algorithms. The "Quantize" algorithm in the ImageMagick software package [3] uses a tree structure to classify and reduce the number of colors in an image. Rather than simply reducing the bit depth as we did in

[3] http://www.imagemagick.org/script/quantize.php

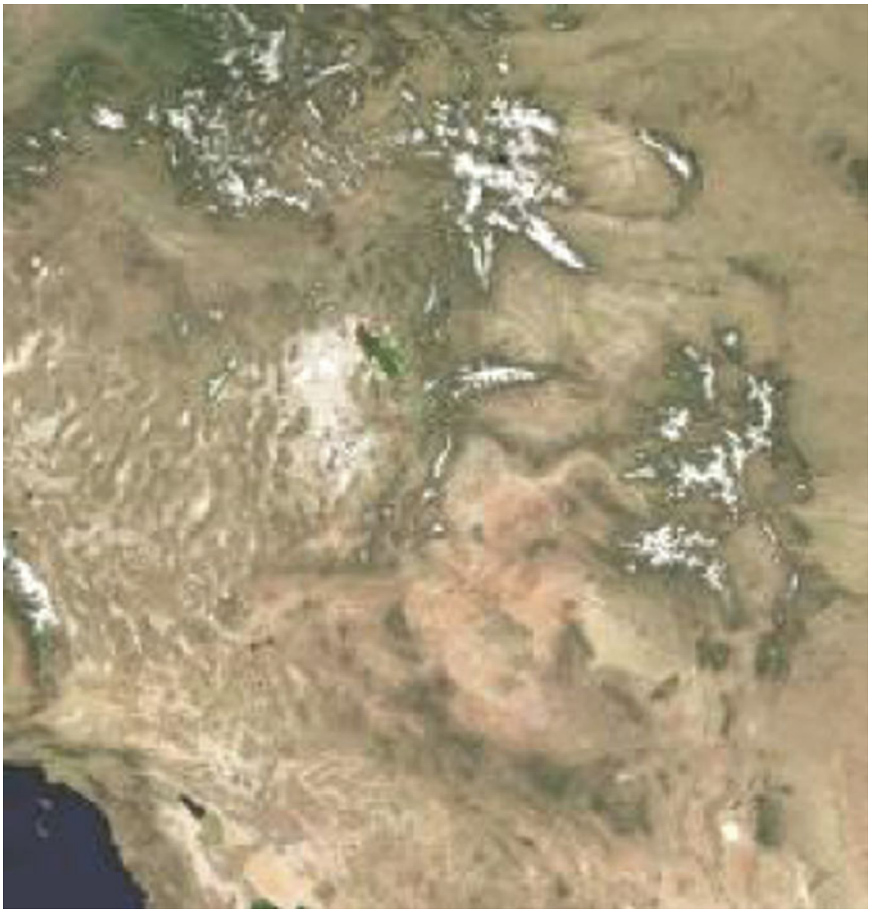
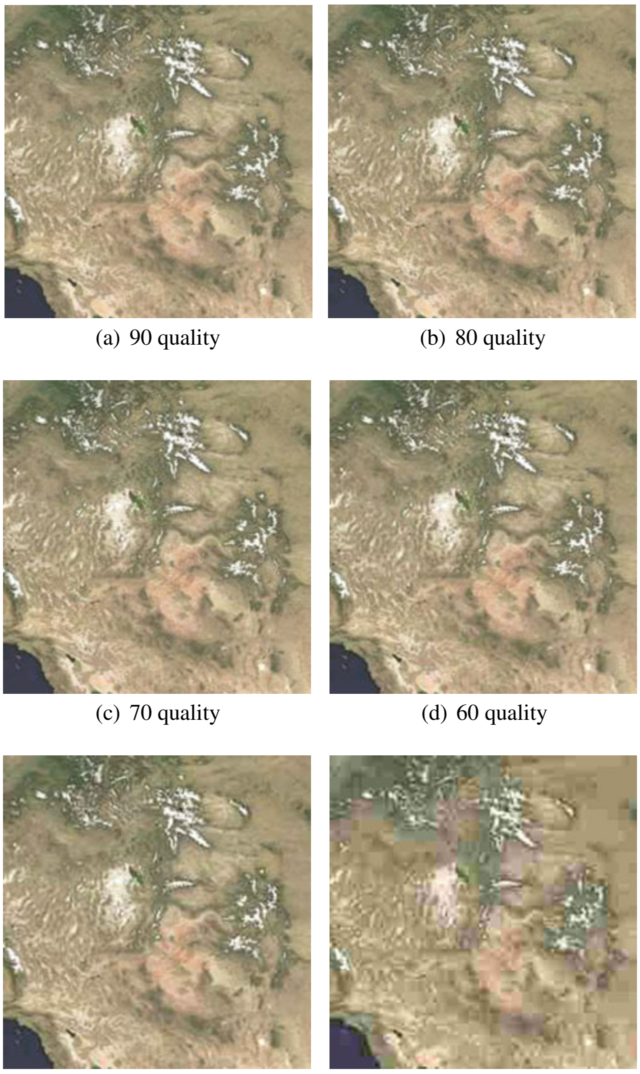


Fig. 4.32 Blue Marble tile image.

RGB Bits	Image Colors	File Size (Bytes)
24	2,372 (Original)	329,662
21	2,094	182,413
18	1,728	176,675
15	1,217	163,290
12	624	148,144
9	191	102,665
6	43	48,660
3	8	26,249

Table 4.9 Results of Simple Color Reduction Algorithm.

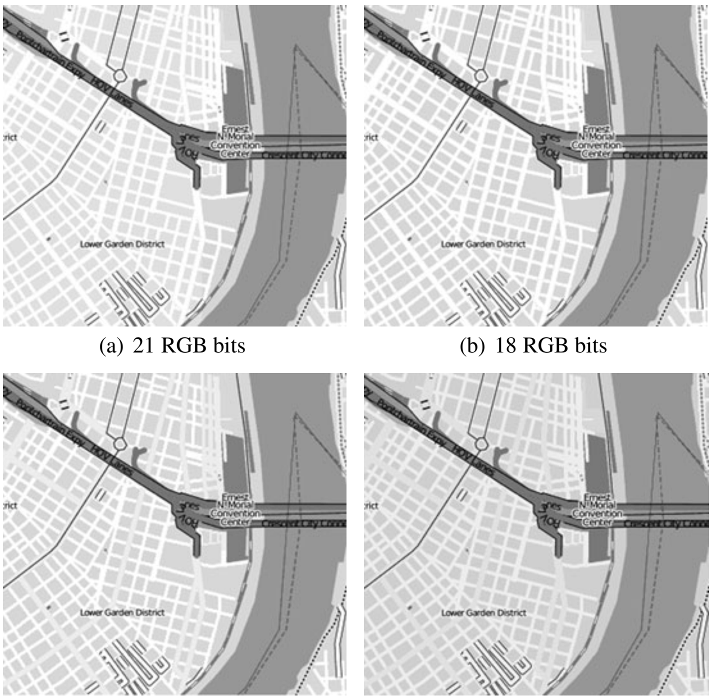


(a) 90 quality (b) 80 quality

(c) 70 quality (d) 60 quality

(e) 40 quality (f) 10 quality

Fig. 4.33 Different quality levels for the same JPEG compressed tile.



(a) 21 RGB bits (b) 18 RGB bits

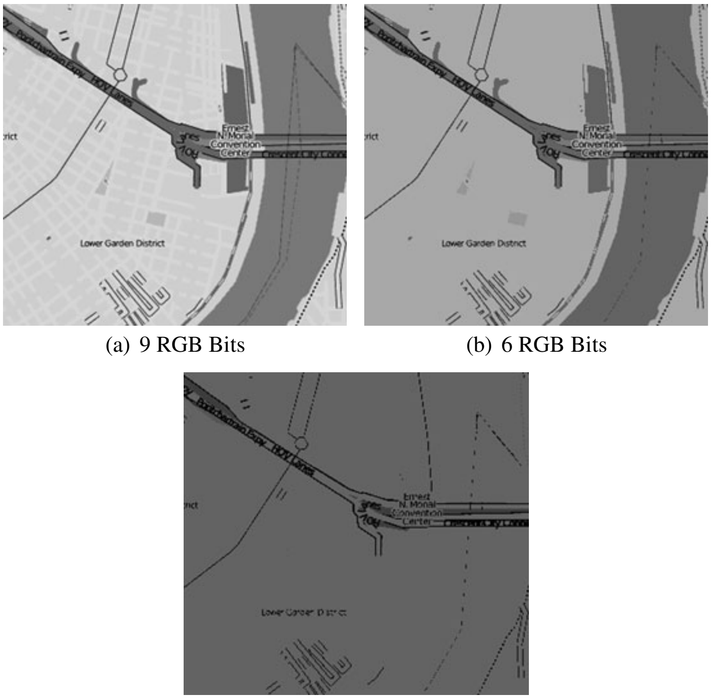
(c) 15 RGB Bits (d) 12 RGB Bits

Fig. 4.34 Comparison between PNG compressed images with differening numbers of bits to represent color. (Part 1)

tour example, this algorithm attempts to intelligently discard colors. It functions by minimizing the color reduction error by calculating the error resulting from the elimination of any single color and then eliminating the colors that have the least impact on the overall error.

Image Colors	File Size (Bytes)
2,372 (Original)	329,662
1659	349,132
798	298,533
400	286,660
100	209,952
50	202,308
16	165,902
8	104,230

Table 4.10 Color reductions using the Quantize algorithm



(a) 9 RGB Bits (b) 6 RGB Bits

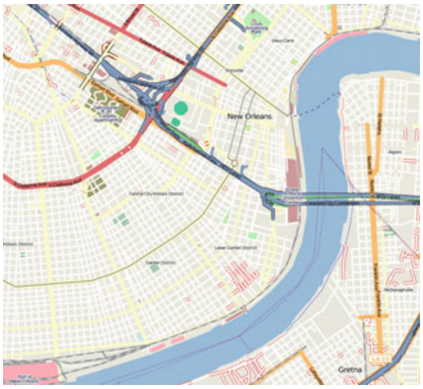
(c) 3 RGB Bits

Fig. 4.35 Comparison between PNG compressed images with different numbers of bits to represent color. (Part 2)

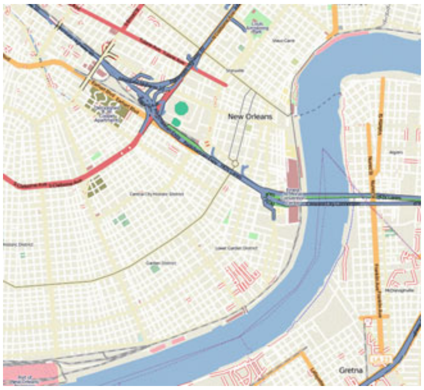
Table 4.10 shows the color reduction to file size reduction numbers. Figures 4.36(a), 4.36(b), 4.36(c), and 4.36(d) show the images reduced to 100, 50, 16, and 8 colors respectively. The images are still usable with as few as 100 colors, but the compression improvements are less significant. Reducing from 2,372 to 100 colors yields only a 36% reduction. We also note the peculiar result that reducing from 2,372 colors to 1,659 actually yields an increase in the file size.

Those producing tiled images will need to conduct their own trials to determine if color reduction is a useful step in their tile production process. We should note that we are reducing the colors of rendered map graphics after they have been rendered. It would be more efficient and more sensible to simply render them with fewer colors from the beginning, if possible.

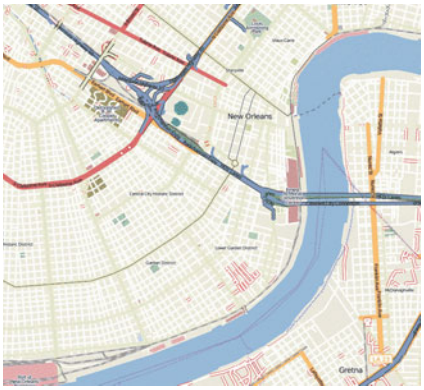
There are other techniques to reduce the sizes of compressed images. For example, we could apply smoothing filters to the images. However, this type of technique is only effective because it reduces the amount of information stored in the image. This also reduces the usefulness of the image.



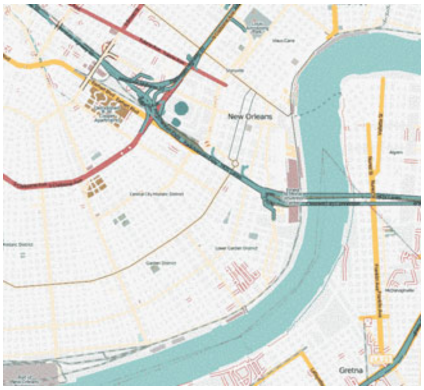
(a) 100 colors



(b) 50 colors



(c) 16 colors



(d) 8 colors

Fig. 4.36 Comparison between PNG compressed images with differening numbers of total colors in the image.

Listing 4.9 Java implementations of nearest neighbor and bilinear interpolation.

```

public class ImageScaling {

    public static void scale(BufferedImage s, BufferedImage t, BoundingBox sbb, BoundingBox tbb) {
        int target_height = t.getHeight();
        int target_width = t.getWidth();
        for (int j = 0; j < target_height; j++) {
            for (int i = 0; i < target_width; i++) {
                Point2DDouble point = geolocate(tbb, i, j, target_height, target_width);
                if (!((point.x > sbb.maxX) || (point.y > sbb.maxY) || (point.x < sbb.minX) || (point.y < sbb.minY))) {
                    int pixelval = interpolate(s, sbb, point);
                    t.setRGB(i, j, pixelval);
                }
            }
        }
    }



    public static Point2DDouble geolocate(BoundingBox coords, int i, int j, int width, int height) {
        double pixel_width = (coords.maxX - coords.minX) / width;
        double pixel_height = (coords.maxY - coords.minY) / height;
        double x = (i + 0.5) * pixel_width + coords.minX;
        int adj_j = height - j - 1;
        double y = (adj_j + 0.5) * pixel_height + coords.minY;
        return new Point2DDouble(x, y);
    }

    private static int interpolate_nn(BufferedImage s, BoundingBox sbb, Point2DDouble point) {
        double tx = point.x;
        double ty = point.y;
        int i = (int) Math.round((tx - sbb.minX) / (sbb.maxX - sbb.minX) * (double) s.getWidth());
        int j = s.getHeight() - 1 - ((int) Math.round((ty - sbb.minY) / (sbb.maxY - sbb.minY) * (double) s.getHeight()));
        return s.getRGB(i, j);
    }

    private static int interpolate_bl(BufferedImage s, BoundingBox sbb, Point2DDouble point) {
        double tx = point.x;
        double ty = point.y;
        int source_height = s.getHeight();
        int source_width = s.getWidth();
        double temp_x = (tx - sbb.minX) / (sbb.maxX - sbb.minX) * source_width;
        double temp_y = source_height - 1 - ((ty - sbb.minY) / (sbb.maxY - sbb.minY) * source_height);
        int i = (int) Math.floor(temp_x);
        int j = (int) Math.floor(temp_y);
        double weight_x = temp_x - i;
        double weight_y = temp_y - j;
        if (j == source_height) {
            j = j - 1;
        }
        if (i == source_width) {
            i = i - 1;
        }
        if (j < 0) {
            j = 0;
        }
        if (i < 0) {
            i = 0;
        }
        int val_0_0 = s.getRGB(i, j);
        int val_0_1 = s.getRGB(i, j);
        int val_1_0 = s.getRGB(i, j);
        int val_1_1 = s.getRGB(i, j);
        int pixel_val_r = getPixelValue(rmask(val_0_0), rmask(val_0_1), rmask(
            val_1_0), rmask(val_1_1), weight_x, weight_y);
        int pixel_val_g = getPixelValue(gmask(val_0_0), gmask(val_0_1), gmask(
            val_1_0), gmask(val_1_1), weight_x, weight_y);
        int pixel_val_b = getPixelValue(bmask(val_0_0), bmask(val_0_1), bmask(
            val_1_0), bmask(val_1_1), weight_x, weight_y);
        int pixel_val = pixel_val_r << 16 | pixel_val_g << 8 | pixel_val_b | 0
            xff000000;
        return pixel_val;
    }

    private static int bmask(int val) {
        int b = val & 0x000000ff;
        return b;
    }

    private static int gmask(int val) {
        int b = (val >> 8) & 0x000000ff;
        return b;
    }

    private static int rmask(int val) {
        int r = (val >> 16) & 0x000000ff;
        return r;
    }

    private static int getPixelValue(int val_0_0, int val_0_1, int val_1_0, int
        val_1_1, double weight_x, double weight_y) {
        int pixel_val = (int) ((1 - weight_x) * (1 - weight_y) * val_0_0 +
            weight_x * (1 - weight_y) * val_0_1 + (1 - weight_x) * (weight_y)
                          * val_1_0 + weight_x * weight_y * val_1_1);
        return pixel_val;
    }

Many programming environments provide built-in tools for scaling and subsetting
    images. This changes our algorithms slightly. Instead of performing
    pixel by pixel calculations, we compute a single set of transformation
    parameters and pass those to the built in image manipulation routines. The
    following code sections show how to use those built in routines in Java
    and Python.

Java Image Scaling and Subsetting


    public static void drawImageToImage(BufferedImage source, BoundingBox
        source_bb, BufferedImage target, BoundingBox target_bb) {
        double xd = target_bb.maxX - target_bb.minX;
        double yd = target_bb.maxY - target_bb.minY;
        double wd = (double) target.getWidth();
        double hd = (double) target.getHeight();
        double targdpx = xd / wd;
        double targdpy = yd / hd;
        double srcdpx = (source_bb.maxX - source_bb.minX) / source.getWidth();
        double srcdpy = (source_bb.maxY - source_bb.minY) / source.getHeight();
        int tx = (int) Math.round(((source_bb.minX - target_bb.minX) / targdpx)
            );
        int ty = target.getHeight() - (int) Math.round(((source_bb.maxY -
            target_bb.minY) / yd) * hd) - 1;
        int tw = (int) Math.ceil(((srcdpx / targdpx) * source.getWidth()));
        int th = (int) Math.ceil(((srcdpy / targdpy) * source.getHeight()));
        Graphics2D target_graphics = (Graphics2D) target.getGraphics();

        //use one of these three statements to set the interpolation method to
             be used
        target_graphics.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
             RenderingHints.VALUE_INTERPOLATION_NEAREST_NEIGHBOR);
        target_graphics.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
             RenderingHints.VALUE_INTERPOLATION_BILINEAR);
        target_graphics.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
             RenderingHints.VALUE_INTERPOLATION_BICUBIC);

        target_graphics.drawImage(source, tx, ty, tw, th, null);
    }
```

Listing 4.10 Java image scaling and subsetting.

```java
public static void drawImageToImage(BufferedImage source,
            BoundingBox source_bb, BufferedImage target,
            BoundingBox target_bb) {
    double xd = target_bb.maxX - target_bb.minX;
    double yd = target_bb.maxY - target_bb.minY;
    double wd = (double) target.getWidth();
    double hd = (double) target.getHeight();
    double targdpx = xd / wd;
    double targdpy = yd / hd;
    double srcdpx = (source_bb.maxX - source_bb.minX) / source.getWidth();
    double srcdpy = (source_bb.maxY - source_bb.minY) / source.getHeight();
    int tx = (int) Math.round(((source_bb.minX - target_bb.minX) / targdpx));
    int ty = target.getHeight() - (int) Math.round(((source_bb.maxY - target_bb
        .minY) / yd) * hd) - 1;
    int tw = (int) Math.ceil(((srcdpx / targdpx) * source.getWidth()));
    int th = (int) Math.ceil(((srcdpy / targdpy) * source.getHeight()));
    Graphics2D target_graphics = (Graphics2D) target.getGraphics();

    //use one of these three statements to set the interpolation method to be
        used
    target_graphics.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
        RenderingHints.VALUE_INTERPOLATION_NEAREST_NEIGHBOR);
    target_graphics.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
        RenderingHints.VALUE_INTERPOLATION_BILINEAR);
    target_graphics.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
        RenderingHints.VALUE_INTERPOLATION_BICUBIC);

    target_graphics.drawImage(source, tx, ty, tw, th, null);
}
```

Listing 4.11 Python image scaling and subsetting.

```python
import Image, ImageDraw # requires the Python Imaging Library (PIL) addon to
    python

def drawImageToImage(source, sourceBoundingBox, target, targetBoundingBox):
    # calculations to determine the degrees per pixel for each dimension of the
        target image
    targetXDelta = targetBoundingBox.maxX - targetBoundingBox.minX
    targetYDelta = targetBoundingBox.maxY - targetBoundingBox.minY
    targetWidth = target.size[0]
    targetHeight = target.size[1]
    targetDegPerPixelX = targetXDelta / float(targetWidth)
    targetDegPerPixelY = targetYDelta / float(targetHeight)

    # calculations to determine the degrees per pixel for each dimension of the
        source image
    # (we collapse the equations into two lines)
    sourceXDelta = sourceBoundingBox.maxX - sourceBoundingBox.minX
    sourceYDelta = sourceBoundingBox.maxY - sourceBoundingBox.minY
    sourceWidth = source.size[0]
    sourceHeight = source.size[1]
    sourceDegPerPixelX = sourceXDelta / float(sourceWidth)
    sourceDegPerPixelY = sourceYDelta / float(sourceHeight)

    targetX = int(round((sourceBoundingBox.minX - targetBoundingBox.minX) /
         targetDegressPerPixelX))
    targetY = targetHeight - int(round(((sourceBoundingBox.maxY -
         targetBoundingBox.minY) / targetYDelta) * targetHeight)) - 1
    tw = int(math.ceil(((sourceDegPerPixelX / targetDegPerPixelX) * sourceWidth
         )))
    th = int(math.ceil(((sourceDegPerPixelY / targetDegPerPixelY) *
         sourceHeight)))

    # use one of these to set the interpolation method when resizing the target
         image
    interpolation = Image.NEAREST
    interpolation = Image.BILINEAR
    interpolation = Image.BICUBIC

    resizedSource = s.resize((tw,th), interpolation)

    im.paste(resizedSource, (targetX, targetY, tw, th))
```

Listing 4.12 Generating randomized map view locations.

```java
public static void tileSizeOptimization1() {
        int numlocations = 10000;
        int[] tilesizes = new int[] {16, 32, 64, 128, 256, 512, 1024, 2048};

        int scale = 10;
        int viewWidth = 1024;
        int viewHeight = 768;

        int numpoints = numlocations;
        Point2DDouble[] randomPoints = getRandomPoints(numpoints);
        int[] totalTilesAccessed = new int[tilesizes.length];
        long[] totalPixelsWasted = new long[tilesizes.length];
        for (int i = 0; i < tilesizes.length; i++) {
            BoundingBox[] bb = getRandomMapViews(randomPoints, scale, tilesizes
                 [i], viewWidth, viewHeight);
            int[] tilesAccessed = getTilesAccessed(bb, scale);
            long[] pixelsWasted = getWastedPixels(tilesAccessed, scale,
                 viewWidth, viewHeight, tilesizes[i]);
            totalTilesAccessed[i] = 0;
            for (int j = 0; j < numlocations; j++) {
                totalTilesAccessed[i] += tilesAccessed[j];
                totalPixelsWasted[i] += pixelsWasted[j];
            }
        }

        for (int i = 0; i < totalTilesAccessed.length; i++) {
            System.out.println(totalTilesAccessed[i] / 10000.0);
        }
        for (int i = 0; i < totalTilesAccessed.length; i++) {
            System.out.println(totalPixelsWasted[i] / 10000.0);
        }

    }

//this method determines the number of pixels that are decoded from the number
     of tiles //accessed and subtracts the number of pixels in the current map
     view
private static long[] getWastedPixels(int[] tiles, int scale, int viewWidth,
     int viewHeight, int tilesize) {
        long[] pixels = new long[tiles.length];
        int pixelsPerTile = tilesize * tilesize;
        int pixelsPerView = viewWidth * viewHeight;
        for (int j = 0; j < pixels.length; j++) {
            pixels[j] = (tiles[j] * pixelsPerTile) - pixelsPerView;
            if (pixels[j] < 0) {
                System.out.println(pixelsPerView + ":" + tiles[j] + ":" +
                    pixelsPerTile);
            }
        }
        return pixels;
    }

    //this calculates the number of tiles that are needed to cover each
        bounding box
    public static int[] getTilesAccessed(BoundingBox[] boxes, int scale) {
        int[] tiles = new int[boxes.length];
        for (int i = 0; i < tiles.length; i++) {
            long mincol = (long) Math.floor(getColForCoord(boxes[i].minX, scale
                ));
            long minrow = (long) Math.floor(getRowForCoord(boxes[i].minY, scale
                ));
            long maxcol = (long) Math.floor(getColForCoord(boxes[i].maxX, scale
                ));
            long maxrow = (long) Math.floor(getRowForCoord(boxes[i].maxY, scale
                ));
            tiles[i] = (int) ((maxcol - mincol + 1) * (maxrow - minrow + 1));
        }
        return tiles;

    }
    //this returns the tile column coordinate that contains the longitude "x"
        for scale "scale"
    static double getColForCoord(double x, int scale) {
        double coord = x + 180.0;
        coord = coord / (360.0 / Math.pow(2.0, (double) scale));
        return (coord);
    }
    //this returns the tile column coordinate that contains the latitude "y"
        for scale "scale"
    static double getRowForCoord(double y, int scale) {
        double coord = y + 90.0;
        coord = coord / (360.0 / Math.pow(2.0, (double) scale));
        return (coord);
    }
    //this computes "numpoints" random x and y locations within our map
        coordinate system
    private static Point2DDouble[] getRandomPoints(int numpoints) {
        Point2DDouble[] points = new Point2DDouble[numpoints];
        for (int i = 0; i < numpoints; i++) {
            double centerX = Math.random() * 360.0 - 180.0;
            double centerY = Math.random() * 180.0 - 90.0;
            Point2DDouble point = new Point2DDouble(centerX, centerY);
            points[i] = point;
        }
        return points;
    }
    //the extrapolates our random x and y locations into map view boxes
    public static BoundingBox[] getRandomMapViews(Point2DDouble[] centerPoints,
         int scale, int tilesize, int viewWidth, int viewHeight) {
        //these two are always the same
        double tileWidthDegrees = 360.0 / Math.pow(2, scale);
        double tileHeightDegrees = 180.0 / Math.pow(2, scale - 1);

        double degreesPerPixel = tileWidthDegrees / tilesize;
        double viewWidthDegrees = viewWidth * degreesPerPixel;
        double viewHeightDegrees = viewHeight * degreesPerPixel;

        BoundingBox[] boxes = new BoundingBox[centerPoints.length];
        for (int i = 0; i < boxes.length; i++) {
            double centerX = centerPoints[i].x;
            double centerY = centerPoints[i].y;
            double minx = centerX - viewWidthDegrees / 2.0;
            double maxx = centerX + viewWidthDegrees / 2.0;
            double miny = centerY - viewHeightDegrees / 2.0;
            double maxy = centerY + viewHeightDegrees / 2.0;
            BoundingBox bb = new BoundingBox(minx, miny, maxx, maxy);
            boxes[i] = bb;
        }
        return boxes;
    }
```

References

1. Denning, P., Schwartz, S.: Properties of the working-set model. Communications of the ACM **15**(3), 198 (1972)
2. Keys, R.: Cubic convolution interpolation for digital image processing. IEEE Transactions on Acoustics, Speech and Signal Processing **29**(6), 1153–1160 (1981)
3. Press, W., Teukolsky, S., Vetterling, W., Flannery, B.: Numerical recipes in C. Cambridge university press Cambridge (1992)