# Chapter 5
# Image Tile Creation

The previous chapter explained the techniques for manipulating geospatial images. This chapter will build on those techniques to explain how a system can be constructed to create sets of tiled geospatial images. In general terms, there are two types of tile generation systems: those that pre-render tiled images and those that render the images in direct response to user queries. Pre-rendering the tiles can require significant processing time, including processing tiles that may never be viewed by users. Rendering tiles just-in-time can save setup time but may require users to wait longer for requested maps. Beyond these differences there are significant technical reasons that usually force us to choose one type of system over the other.

Systems that serve tiled geospatial images from rendered vector content almost always use a form of just-in-time tiling. There are three reasons for this:

- Storage space: Rendered image tiles require a significant amount of storage space relative to vector map content. A collection of geospatial features might be 100 megabytes in vector form but could grow to several terabytes when rendered over several different levels.
- Processing time: Pre-rendering image tiles requires a significant amount of time, and many of those tiles may be in geographic areas of little interest to users. The most efficient method of deciding what tiles to render is to wait until they are requested by actual users.
- Overview images: Overview images, i.e., very low zoom level images, can be rendered directly from geospatial vectors. Unlike raster based tile systems, there is no need to render the high level views first and then generate scaled down versions.

Conversely, tiling systems that primarily draw from sets of geospatial imagery typically pre-render all image tiles. There are two reasons for this:

- Processing time: Reformatting, scaling, and reprojecting of imagery are often required in the tile creation process. These steps can be too time consuming for users to wait for in real time.

J.T. Sample and E. Ioup, *Tile-Based Geospatial Information Systems: Principles and Practices*, DOI 10.1007/978-1-4419-7631-4_5,
© Springer Science+Business Media, LLC 2010

- Overview images: Low level images must be created from higher resolution versions of the same imagery. This requires the higher levels to be completely rendered before the low resolution levels can be completed.

Since the primary focus of this book is tiling systems based on imagery, this chapter will examine tiling systems that pre-render image tiles. Chapter 11 will discuss tiling systems based on vector geospatial data.

## 5.1 Tile Creation from Random Images

Tiled image sets are created from collections of random source images. We call the source images random because, unlike our tiled images, the source images may have sizes and boundaries that follow no specific system. Collections of source images come in varied forms. For example, we might have high-resolution aerial imagery of the world's 50 largest cities. Each city is represented by a small number (5 to 50) of large source images (approximately 10,000 by 10,000 pixels). Each of the source images can have a different size, cover a different portion of the earth, or have a different map resolution. Taken together, all of the source images for all of the cities form a single map layer. Common file formats for source images include GEOTIFF, MrSID, and GEO-JPEG2000. However, almost any image format can be used, as long as the geospatial properties of the image are encoded either in the image file or along side it.

At this point it is useful to define the concept of a map layer. Layers are typically the atomic unit for requesting map data from web-based geospatial systems. A map layer is a logical grouping of geospatial information. The term ”layer” is used to convey the idea of multiple graphical layers stacked in some order in a visual display. Layers are formed by logical groupings of geospatial data. For example, an ”entertainment” map layer would include locations of movie theaters, parks, zoos, museums, etc. In the case of imagery, we will formally define a layer as a single map view for a given geographic area.

**Corallary 1** *For a single layer, there exists one and only one tile image for a specific tile address.*

For example, consider that we have aerial imagery over Alaska. The snow cover over Alaska varies from month to month. We might have source image sets for every month of the year. Each of those image sets covers the same area. For a given tile address, each image set would have a tiled image with different content. Therefore, we have to group them into separate layers, one layer for the image set for each month.

## 5.2 Tile Creation Preliminaries

For the purposes of this chapter, we will consider the problem of taking a set of random source images and converting them into a layer of tiled images. Since a set of random source images will have images of varying size and resolution, the tile creation process is simply the process of scaling and shaping image data from the source images into tile-sized pieces.

### *5.2.1 Bottom-Up Tile Creation*

Each layer of tile images has multiple levels. A tile set starts with a base level; the base level is the level with the highest number and the highest resolution imagery. Each subsequent level is a lower version of the level preceding it. Figure 5.1 shows three levels of the same image layer. In this example, Level 3 is the base level. Levels 2 and 1 are lower resolution versions of the same data. The images in Figure 5.1 show the tile boundaries according to the logical tile scheme presented in Chapter 2.

**Definition 1** *The base level for a tile layer is the highest resolution level, the level with the highest number for that tile layer.*

In the tile creation process, the base level is almost always completed, at least partially, before lower resolution levels. Therefore, we can say that tile creation is a bottom-up process in terms of map scale.

### *5.2.2 Choosing the Base Level for a Set of Source Images*

Before a tile set can be created from a set of random images, the base level must be chosen. In some cases, a target base level is determined ahead of time. It could be required to integrate with other tile layers or client software. However, in most cases the base level is chosen to closely match the resolution of the sources images. A given set of random source images is unlikely to exactly match one of our predetermined level resolutions. So we must select the tile level that most closely matches our source images. Table 5.1 shows the first 19 levels in our logical tile scheme with 512 by 512 pixel tiles. It gives the number of horizontal and vertical tiles for each level along with each level's degrees per pixel (DPP). These are calculated using the formulas provided in Chapter 2. The DPP values will be used to choose base levels for sets of source images.

To perform this analysis, we will require some information from each of the source images in our set: image width and image height in pixels and minimum and maximum vertical and horizontal coordinates in degrees. As shown in Listing 5.1, we can compute the DPP value for our set of random images. We compute the DPP

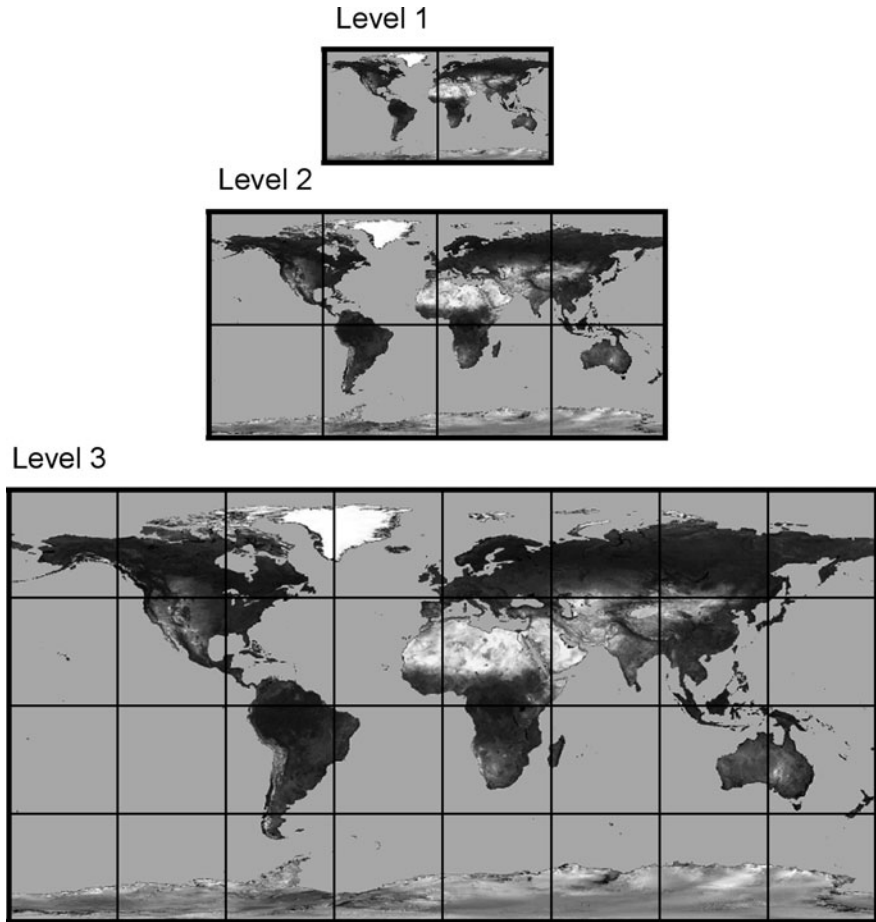


**Fig. 5.1** Multiple zoom levels of the same layer.

value by combining the vertical and horizontal dimensions of the images. This is a valid procedure for tiled image projections that preserve the same DPP in each dimension as our logical tile scheme does. The calculations are performed in degrees. Source images stored in other projections might use meters with its coordinate system. In those cases, conversion to degrees is required. Suppose that for a set of source images, we have computed a DPP value of 0.03. This falls in between levels 4 and 5. If we choose level 4, we will be scaling DOWN our source images and thus losing a little bit of data from the source images. If we choose level 5, we will be scaling UP our source images. We will preserve all the data but take up more storage space.

For example, if our source image set takes up 10,000,000 bytes uncompressed with a DPP value of 0.03, when converted to level 4 it will take up 4,660,000 bytes,

| Level | Horizontal Tiles | Vertical Tiles | Degrees Per Pixel |
|---|---|---|---|
| 1 | 2 | 1 | 0.3515625 |
| 2 | 4 | 2 | 0.17578125 |
| 3 | 8 | 4 | 0.087890625 |
| 4 | 16 | 8 | 0.0439453125 |
| 5 | 32 | 16 | 0.02197265625 |
| 6 | 64 | 32 | 0.010986328125 |
| 7 | 128 | 64 | 0.0054931640625 |
| 8 | 256 | 128 | 0.00274658203125 |
| 9 | 512 | 256 | 0.001373291015625 |
| 10 | 1024 | 512 | 0.000686645507812 |
| 11 | 2048 | 1024 | 0.000343322753906 |
| 12 | 4096 | 2048 | 0.000171661376953 |
| 13 | 8192 | 4096 | 0.000085830688477 |
| 14 | 16384 | 8192 | 0.000042915344238 |
| 15 | 32768 | 16384 | 0.000021457672119 |
| 16 | 65536 | 32768 | 0.00001072883606 |
| 17 | 131072 | 65536 | 0.00000536441803 |
| 18 | 262144 | 131072 | 0.000002682209015 |
| 19 | 524288 | 262144 | 0.000001341104507 |

**Table 5.1** Number of tiles and degrees per pixel for each level.

**Listing 5.1** Computation of degrees per pixel for a set of random source images.

```
1  ddpX = 0.0
2  ddpY = 0.0
3  count = 0
4
5  for image in images:
6      count = count + 1
7      ddpX = ddpX + (image.maxX − image.minX)/image.width
8      ddpY = ddpY + (image.maxY − image.minY)/image.height
9
10 dppTotal = (dppX + dppY)/(2∗count)
```

a reduction of 53%. When converted to level 5 it will take up 18,645,000 bytes, an increase of 86%, or nearly double the original amount.

Equation 5.1 gives an approximate computation of the storage space changes affected by transforming from the native level to a fixed tile level, where $N$ is the native resolution in degrees per pixel, $B$ is the base level resolution in degrees per pixel, $S$ is the size of the source image, and $R$ is the space required for the tiled image set. The exact storage space changes cannot be calculated analytically. There are several unknown factors, such as the impact of uneven source image breaks onto the tile boundaries. Since all images are stored in a compressed format, the exact storage space requirements can be calculated only by creating and compressing the images.

$$R = (\frac{N}{B}) * 2S \tag{5.1}$$

In general, we want to choose the level with the closest DPP value that is lower than our native DPP as our base level. This will usually result in an increase of storage space requirements, but it will preserve the image information. Practically speaking, geospatial image data costs much more to create than to store. Satellites, aerial platforms, and cartographers are all more expensive than hard drives.

Since the tiled and source images are compressed, the actual increase in storage space requirements is usually smaller than Equation 5.1 would predict. Image compression algorithms attempt to compress images by storing just the information needed to reproduce the image. Since our rescaled tiled images are not adding any real image information, we can expect the compressed results to be similar in size to the original. Consider an example: the NASA Blue Marble image below is 2000 by 1000 pixels in size (Figure 5.2). Compressed as a JPEG image it is 201,929 bytes. If we resize the image to 3000 by 1500, we have increased the images number of pixels by a ratio of 2.25. However, the new larger image compressed as a JPEG takes up 360,833 bytes, a growth ratio of 1.79. So, the actual storage space requirements grew by 79%, not 125% as predicted by simple pixel calculations. Table 5.2 shows these results. It includes results for another scaled image that further illustrate the principle.

Once we have chosen and created the tiled images for the base level, the lower resolution levels can be created.

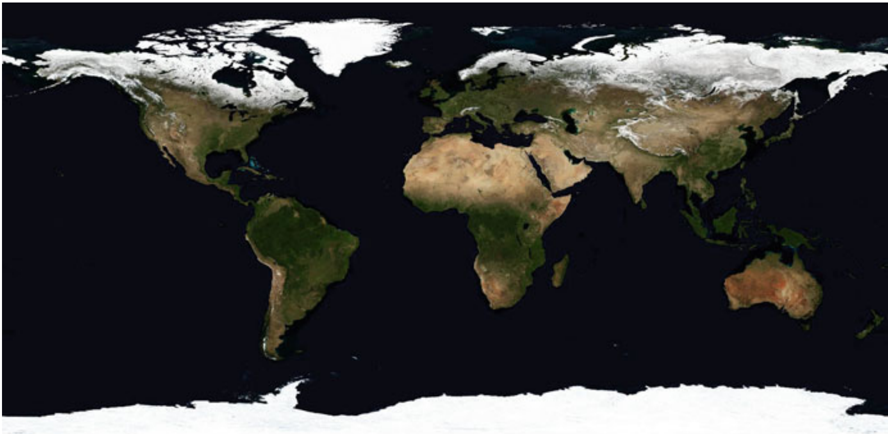

**Fig. 5.2** 2000 by 1000 Blue Marble image.

| | Width | Height | Total Pixels | Percent Increased | Compressed Size | Percent Increased |
|---|---|---|---|---|---|---|
| Original Image | 2000 | 1000 | 2,000,000 | | 201,929 | |
| Scaled Image 1 | 3000 | 1500 | 4,500,000 | 125% | 360,833 | 79% |
| Scaled Image 2 | 5000 | 2500 | 12,500,000 | 525% | 816,446 | 304% |

**Table 5.2** Compression ratios are greater for the same image at different resolutions.

**Listing 5.2** Pull-based tile creation.

```
1 for t in tiles:
2     for s in sources:
3         if s.intersects(t):
4             p = extractPixels(s)
5             drawPixels(p,t)
```

**Listing 5.3** Push-based tile creation.

```
1 for s in sources:
2     for t in tiles:
3         if t.intersect(s):
4             p = extractPixels(s)
5             drawPixels(p,t)
```

### *5.2.3 Pull-Based Versus Push-Based Tile Creation*

There are two methods for creating the tiles from random source images: pull-based and push-based. Pull-based tile creation iterates over the desired tiles and pulls image data from the source images. Push-based tile creation iterates over the source images and pushes image data from them to the tiled images. There is little difference between these two approaches. The following pseudo code example shows that only the ordering of the iteration structure changes between the two methods, as shown in Listings 5.2 and 5.3.

In practice, there are several technical concerns that make the two methods substantially different. First and foremost is the issue of memory. If our computers had infinite memory, and all source and tile images could be held completely in memory, then there would be no effective difference between the two approaches. However, computers have limited memory, and we must move our source and tile images in and out of memory as we use them. Reading and decoding compressed source images from disk can be time consuming, as is compressing and writing tiled images to disk.

A second concern is that of multi-threading. Modern computers have multiple processing cores and can execute multiple threads simultaneously. A practical system must make use of multiple threads to be efficient, but it must also be careful to manage image resources in a thread safe fashion. Two threads should probably not operate on the same image tile at the same time.

For our first prototype tile creation system, we will use a pull-based method. Many of the tiles will contain data from multiple source images. If we iterate over source images first, as in a push-based method, then we will be swapping tiled images in and out of memory often. So those tiles will have to be swapped between memory and disk multiple times in the process of creating them. This poses several problems when it comes to tile storage, as many writes of small files or data blocks tends to cause fragmentation of a file system or database pages. In the next chapter we will discuss in more detail why it is important to write an image tile once and only once.

Unlike tiled images, our source images are used in a read-only fashion. We can safely swap them in and out of memory many times without having to perform any writes. This leads us to use a pull-based method. We will iterate over the tiles first and swap the large source images in and out of memory. This result may seem counterintuitive. Typically our source images are much larger than our tiled images. Sources images commonly range from 1,000 by 1,000 to 10,000 by 10,000. Our tile images are either 256 by 256 or 512 by 512. The large source images will take a significant amount of time to read and re-read from disk.

To mitigate this result, we will use a memory cache of source images. We will construct a Least Recently Used (LRU) cache of decoded source images in memory. LRU caches have a fixed size. If an element is added to an already full cache, the LRU cache will discard the least recently used element. Each time we access a source image, we will check if it is in the memory cache. If it is, then we do not have to read and decode the image. If the image is not in the memory cache, we will read and decode the image and place it in the cache.

The LRU cache works very well in this case. We will iterate over tiles in geographic order. Source images affect groups of tiles that border each other geographically. We can expect to have a high rate of ”hits” on our memory image cache. The first tile that requests data from a source image will cause it to be loaded in the cache. The tiles immediately following the first tile will probably also use data from that source image which was just placed in the cache. This high cache hit rate provides a more efficient algorithm.

## 5.3 Tile Creation Algorithms

The following are the steps in the tile creation process:

1. Choose the base level for the tile set.
2. Determine the geographic bounds of the tile set. (This can be based on the bounds of the source images.)
3. Determine the bounds of the tile set in tile coordinates.
4. Initialize the tile storage mechanism.
5. Iterate over the tile set coordinates. For each tile, do the following:

    a. Compute the geographic bounds of the specific tile.

   b. Iterate over the source images. For each source image do the following:
      i. Determine if the specific source image intersects the tile being created.
      ii. If the source image and tile intersect,
         A. Check the cache for the source image. If it is not in the cache, load it from disk and save in the cache.
         B. Extract the required image data from the source image, and store it in the tiled image.
   c. Save the completed tiled image to the tile storage mechanism.

6. Clear the source image cache.
7. Finalize the tile storage mechanism.

Before presenting the computer code for executing these steps, we will define the following data types in Listing 5.5:

| | |
|---|---|
| BoundingBox: | Wrapper for bounding rectangle in degrees. |
| SourceImage: | Wrapper for image dimensions and geographic bounds. |
| TileAddress: | Wrapper for a tile's row, column, and level coordinates. |
| BufferedImage: | Built-in Java class for memory images. |
| TileOutputStream: | Abstract class for output of tiled images. |
| MemoryImageCache: | Abstract class for a LRU cache of source images. |

There are several key methods embedded in these data types. `TileAddress.getBoundingBox()` provides the bounding coordinates in degrees for an image tile address. `BoundingBox.intersects()` tests if two bounding boxes intersect each other. `BoundingBox.union()` is used to combine multiple bounding boxes into a single one. The abstract method `writeTile()` is used to provide a generic means for storing tiles. Concrete implementations of this will be discussed in the next chapter. Additional abstract methods, `getSourceImageData()` and `putSourceImageData()`, are used to provide access to the LRU source image cache. Implementation of this is left to the reader. We will also use the function `drawImageToImage`, which was defined in the previous chapter. Formulae for computing tile and geographic coordinates are derived in Chapter 2. We will use the constant `TILE_SIZE` to represent the width and height of our tiled images. This value is the same for the horizontal and vertical dimensions. See the previous chapter for a thorough discussion of how to choose the best tile size. Listing 5.6 is Java code for a basic, single threaded method for creating the base level of a tile set.

### *5.3.1 Scaling Process for Lower Resolution Levels*

The previous algorithm created the base level. Next we create the lower resolution levels. Each lower level is based on the previous level. Because of the structured nature of our logical tile scheme, this process is much simpler than creation of the base level. Figure 5.3 shows that our lower resolution tiles are constructed directly and from exactly fours tiles from the previous level.

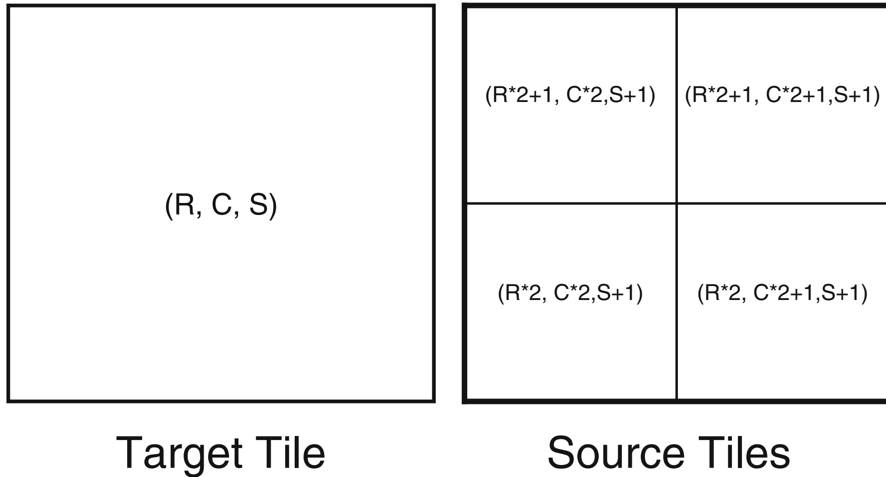


**Fig. 5.3** Relationship of target tile for source tiles from previously computed level.

The basic algorithm is as follows:

1. For each level from (`base_level` - 1) to 1, do the following.
   a. Determine the bounds of the current tile level in tile coordinates.
   b. Initialize the tile storage mechanism.
   c. Iterate over the tile set coordinates. For each tile, do the following:
      i. Determine the four tiles from the higher level that contribute to the current tile.
      ii. Retrieve the four tile images or as many as exist.
      iii. Combine the four tile images into a single, scaled-down image.
      iv. Save the completed tiled image to the tile storage mechanism.
   d. Finalize the tile storage mechanism.

This algorithm uses the types defined in the previous section plus one additional type as defined in Listing 5.6. This type allows us to read the tiles from the previous levels. Additionally, an assumption is made that the TileInputStream and TileOutputStream in the algorithm are linked in some fashion. This allows us to write tiles in one stage and then read them out in the next stage. For example, when creating level 7, we will write level 7 tiles to the TileOutputStream. In the next stage, when we create level 6, we will have to read the level 7 tiles that we created in the previous step. Listing 5.7 shows the complete algorithm for creating the lower resolution layers from the base layer.

The process of creating tiled image sets from collections of random source images can be approached in a straightforward manner. In this chapter, we have detailed the basic process and algorithms for achieving this goal. We have built upon

**Listing 5.4** Abstract class definition for the TiledInputStream.

```
1 abstract class TileInputStream {
2     abstract BufferedImage getTile(TileAddress address);
3 }
```

the image manipulation algorithms from previous sections. In the next chapter, we will present techniques for optimizing the creation of tiled image sets.

**Listing 5.5** Java data types.

```
1  class BoundingBox {
2
3      double minx, miny, maxx, maxy;
4
5      public BoundingBox(double minx, double miny, double maxx, double maxy) {
6          this.maxx = maxx;
7          this.maxy = maxy;
8          this.minx = minx;
9          this.miny = miny;
10     }
11
12     boolean intersects(double minx, double miny, double maxx, double maxy) {
13         return !(minx > this.maxx || maxx < this.minx || miny > this.maxy ||
               maxy < this.miny);
14     }
15
16     static BoundingBox union(BoundingBox[] bb) {
17         BoundingBox u = bb[0];
18         for (int i = 1; i < bb.length; i++) {
19             if (bb[i].maxx > u.maxx) {
20                 u.maxx = bb[i].maxx;
21             }
22             if (bb[i].maxy > u.maxy) {
23                 u.maxy = bb[i].maxy;
24             }
25             if (bb[i].minx < u.minx) {
26                 u.minx = bb[i].minx;
27             }
28             if (bb[i].miny < u.miny) {
29                 u.miny = bb[i].miny;
30             }
31         }
32         return u;
33     }
34 }
35
36 class SourceImage {
37
38     int width;
39     int height;
40     BoundingBox bb;
41     BufferedImage image;
42     String name;
43 }
44
45 class TileAddress {
46
47     long row;
48     long column;
49     int scale;
50
51     BoundingBox getBoundingBox() {
52         double dp = 360.0 / (Math.pow(2, scale) * TILE_SIZE);
53         double miny = (row * TILE_SIZE * dp) - 90.0;
54         double maxy = ((row + 1) * TILE_SIZE * dp) - 90.0;
55         double minx = (column * TILE_SIZE * dp) - 180.0;
56         double maxx = ((column + 1) * TILE_SIZE * dp) - 180.0;
57         BoundingBox bb = new BoundingBox(minx, miny, maxx, maxy);
58         return bb;
59     }
60 }
61
62 abstract class TileOutputStream {
63
64     abstract void writeTile(TileAddress address, BufferedImage image);
65
```

```
66 }
67
68 abstract class MemoryImageCache {
69
70     abstract BufferedImage getSourceImageData(String name);
71
72     abstract void putSourceImageData(String name, BufferedImage data);
73 }
```

**Listing 5.6** Simple tile creation.

```
1  public void createTiles(SourceImage[] sourceImages, TileOutputStream
       tileOutputStream, int baseLevel, MemoryImageCache cache) {
2
3      //Determine the geographic bounds of the tile set.
4      //This can be based on the bounds of the source images.
5      BoundingBox[] sourceImageBounds = new BoundingBox[sourceImages.length];
6      for (int i = 0; i < sourceImageBounds.length; i++) {
7          sourceImageBounds[i] = sourceImages[i].bb;
8      }
9      BoundingBox tileSetBounds = BoundingBox.union(sourceImageBounds);
10     //Determine the bounds of the tile set in tile coordinates.
11     long mincol = (long) Math.floor((tileSetBounds.minx + 180.0) / (360.0 /
           Math.pow(2.0, (double) baseLevel)));
12     long maxcol = (long) Math.floor((tileSetBounds.maxx + 180.0) / (360.0 /
           Math.pow(2.0, (double) baseLevel)));
13     long minrow = (long) Math.floor((tileSetBounds.miny + 90.0) / (180.0 / Math
           .pow(2.0, (double) baseLevel - 1)));
14     long maxrow = (long) Math.floor((tileSetBounds.maxy + 90.0) / (180.0 / Math
           .pow(2.0, (double) baseLevel - 1)));
15
16     //Iterate over the tile set coordinates.
17     for (long c = mincol; c <= maxcol; c++) {
18         for (long r = minrow; r <= maxrow; r++) {
19             TileAddress address = new TileAddress(r, c, baseLevel);
20             //Compute the geographic bounds of the specific tile.
21             BoundingBox tileBounds = address.getBoundingBox();
22             //Iterate over the source images.
23             BufferedImage tileImage = new BufferedImage(TILE_SIZE, TILE_SIZE,
                   BufferedImage.TYPE_INT_ARGB);
24             for (int i = 0; i < sourceImages.length; i++) {
25                 //Determine if the specific source image intersects the tile
                       being created.
26                 if (sourceImages[i].bb.intersects(tileBounds.minx, tileBounds.
                       miny, tileBounds.maxx, tileBounds.maxy)) {
27                     //Check the cache for the source image.
28                     BufferedImage bi = cache.getSourceImageData(sourceImages[i
                           ].name);
29                     if (bi == null) {
30                         //If it is not in the cache load it from disk and save
                               in the cache.
31                         bi = readImage(sourceImages[i].name);
32                         cache.putSourceImageData(sourceImages[i].name, bi);
33                     }
34                     //Extract the required image data from the source image and
                           store it in the tiled image.
35                     drawImageToImage(bi, sourceImages[i].bb, tileImage,
                           tileBounds);
36                 }
37             }
38             //Save the completed tiled image to the tile storage mechanism.
39             tileOutputStream.writeTile(address, tileImage);
40         }
41     }
42 }
```

**Listing 5.7** Scaled tile creation.

```
1  public void createScaledTile(TileInputStream tileInputStream, TileOutputStream
       tileOutputStream, int baseLevel, long minCol, long maxCol,
2          long minRow, long maxRow) {
3      //For each level from base level - 1 to 1, do the following.
4      for (int level = baseLevel - 1; level <= 1; level--) {
5          //Determine the bounds of the current tile level in tile coordinates.
6          //ratio will be used to reduce the original tile set bounding
               coordinates to those applicable for each successive level.
7          int ratio = (int) Math.pow(2, baseLevel - level);
8          long curMinCol = (long) Math.floor(minCol / ratio);
9          long curMaxCol = (long) Math.floor(maxCol / ratio);
10         long curMinRow = (long) Math.floor(minRow / ratio);
11         long curMaxRow = (long) Math.floor(maxRow / ratio);
12         //Iterate over the tile set coordinates.
13         for (long c = curMinCol; c <= curMaxCol; c++) {
14             for (long r = curMinRow; r <= curMaxRow; r++) {
15                 //For each tile, do the following:
16                 TileAddress address = new TileAddress(r, c, level);
17                 //Determine the FOUR tiles from the higher level that
                       contribute to the current tile.
18                 TileAddress tile00 = new TileAddress(r * 2, c * 2, level + 1);
19                 TileAddress tile01 = new TileAddress(r * 2, c * 2, level + 1);
20                 TileAddress tile10 = new TileAddress(r * 2, c * 2, level + 1);
21                 TileAddress tile11 = new TileAddress(r * 2, c * 2, level + 1);
22                 //Retrieve the four tile images, or as many as exist.
23                 BufferedImage image00 = tileInputStream.getTile(tile00);
24                 BufferedImage image01 = tileInputStream.getTile(tile01);
25                 BufferedImage image10 = tileInputStream.getTile(tile10);
26                 BufferedImage image11 = tileInputStream.getTile(tile11);
27                 //Combine the four tile images into a single, scaled-down image
                       .
28                 BufferedImage tileImage = new BufferedImage(TILE_SIZE,
                       TILE_SIZE, BufferedImage.TYPE_INT_ARGB);
29                 Graphics2D g = (Graphics2D) tileImage.getGraphics();
30                 g.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
                       RenderingHints.VALUE_INTERPOLATION_BILINEAR);
31                 boolean hadImage = false;
32                 if ((image00 != null)) {
33                     g.drawImage(image00, 0, Constants.TILE_SIZE_HALF, Constants
                           .TILE_SIZE_HALF, Constants.TILE_SIZE, 0, 0, Constants.
                           TILE_SIZE,
34                             Constants.TILE_SIZE, null);
35                     hadImage = true;
36                 }
37                 if ((image10 != null)) {
38                     g.drawImage(image10, Constants.TILE_SIZE_HALF, Constants.
                           TILE_SIZE_HALF, Constants.TILE_SIZE, Constants.
                           TILE_SIZE, 0, 0,
39                             Constants.TILE_SIZE, Constants.TILE_SIZE, null);
40                     hadImage = true;
41                 }
42                 if ((image01 != null)) {
43                     g.drawImage(image01, 0, 0, Constants.TILE_SIZE_HALF,
                           Constants.TILE_SIZE_HALF, 0, 0, Constants.TILE_SIZE,
44                             Constants.TILE_SIZE, null);
45                     hadImage = true;
46                 }
47                 if ((image11 != null)) {
48                     g.drawImage(image11, Constants.TILE_SIZE_HALF, 0, Constants
                           .TILE_SIZE, Constants.TILE_SIZE_HALF, 0, 0, Constants.
                           TILE_SIZE,
49                             Constants.TILE_SIZE, null);
50                     hadImage = true;
51                 }
52                 //save the completed tiled image to the tile storage mechanism.
53                 if (hadImage) {
```

```
54                         tileOutputStream.writeTile(address, tileImage);
55                     }
56                 }
57             }
58         }
59 }
```