Chapter 12
Case Study: Tiles from Blue Marble Imagery

In this chapter we will present a complete end-to-end system for creating and storing tiled images from a freely available worldwide set of imagery. The system will read source imagery, cut it into tiled images, and store the tiled images to cluster files.

NASA's Blue Marble Next Generation Imagery (BMNG) is a composite image of the Earth at 500 meters resolution taken by the MODIS satellite mounted sensor. The BMNG imagery and information about it are freely available for download from

`http://earthobservatory.nasa.gov/Features/BlueMarble/`

The imagery comes in two formats: as a single raw image file 86,400 pixels wide by 43,200 pixels high and as 8 smaller sub-images, 21,600 pixels by 21,600 pixels. In this chapter we will present a pull-based tiling approach using the single large image and a push-based tiling approach using the 8 sub-images.

Before we can begin tiling, we must determine the base zoom level that we will use for our tile set. Both the single large image and the set of 8 sub-images have the same geospatial and image resolution, so we will use the same base zoom level for both image sets. Using the following equation, we can compute the degrees per pixel for our Blue Marble imagery.

$$(360.0/84600 + 180.0/42300)/2 = 0.00425.$$

Since 0.00425 falls between level 7 (0.00549) and level 8 (0.00274), as shown in Table 2.1, we will choose level 8 as our base level.

12.1 Pull-Based Tiling

The algorithm presented in this section will bring together six concepts already presented in the book:

J.T. Sample and E. Ioup, *Tile-Based Geospatial Information Systems: Principles and Practices*, DOI 10.1007/978-1-4419-7631-4_12,

- Section 5.2.3: Pull-based tile creation that iterates over the tiles first. For each tile, it extracts the required data from the source images, creates the tile, and stores the tile.
- Section 5.3.1: Scaling process for lower resolution tiles.
- Section 6.1: An optimized version of tile creation that holds tiles in memory while they are being used and write them to disk in memory.
- Section 6.2.1: Methods for partial reading of source images.
- Section 6.3.1: Multi-threading tile creation.
- Section 8.5: Storage of tiled images in clusters of tiles from different zoom levels.

Because our source image is too large to hold in memory all at once, we will implement an algorithm for partial reading of the image. The image, like many images, is stored in row-major order, also known as scanline order. The Java class in Listing 12.1 provides a method for reading pixel delineated sub-sections of the large Blue Marble image. The example code is designed for clarity and ease of understanding. There are more efficient ways to read sub-images. These include setting pixels in blocks of data versus setting one pixel at a time and reading blocks of bytes instead of one byte at a time. For simplicity's sake, we will multi-thread the creation of only the base level. The higher levels take a much shorter amount of time to create and do not require multi-threading.

The next piece of supporting code we will need is a modified version of the tile cluster storage algorithm. The version in Listing 12.2 takes the code presented in Section 8.5 and adds in-memory caching of tiled images during the creation process and a cache of open RandomAccessFiles. Since this section is primarily concerned with creating tiles, the code only manages a write cache. Reading of tiles stored to disk in an earlier session is always done directly from disk. Reading of tiles that have just been written to the cache is done from the cache. Also, the write cache uses a hashmap with Java String objects as keys. A more efficient approach would use numerical tile addresses as keys, but the String based approach is simpler to implement. The final piece of supporting code, Listing 12.3, allows multiple threads to iterate over a range of tile addresses.

Given the supporting code, now we can create the completed system. The steps in the algorithm are as follows:

1. Iterate over all the tiles in the base zoom level. For each tile:
   a. Pull the imagery needed from the source image.
   b. Scale the tile to the proper resolution.
   c. Store the tile in the cache.
2. Iterate over each successive zoom level up to level 1.
   a. For each zoom level, iterate over each tile at that level. For each tile:
      i. Pull the four images from the higher level that make up the current tile.
      ii. Merge the four images together into one image.
      iii. Store that image into cache.
3. After all tiles have been created, write the tiles to disk.

The Java classes in Listing 12.4 demonstrate this algorithm. The first class, PullTileCreation, initializes the input and output, creates and starts the pull tiler threads, and creates the lower resolution levels. The class PullTilerThread is a Java thread implementation that does the work of creating the base level tiled images.

12.2 Push-Based Tiling

In this example, we will use as source imagery the Blue Marble data that has been divided into 8 sub-images. Each image is 21,600 by 21,600 pixels and covers a 90 degree by 90 degree area of the earth's surface. Since each sub-image can be held completely in memory, we can use a push-based tiling approach. Our algorithm will iterate over the source images in memory and extract data from each source image needed to make the tiled images.

The algorithm presented in this section will bring together four concepts already presented in the book:

- Section 5.2.3: Push-based tile creation that iterates over the source images first.
- Section 5.3.1: Scaling process for lower resolution tiles.
- Section 6.1: An optimized version of tile creation that held tiles in memory while they were being used and wrote them to disk in memory.
- Section 8.5: Storage of tiled images in clusters of tiles from different zoom levels.

It should be noted that the two techniques differ only in the method of creating the base level tiles. The section of code for creating the lower resolution levels is exactly the same as in the previous section. Listing 12.5 shows the push-based method for creating the Blue Marble tiles.

12.3 Results

Each technique creates the required tile clusters, and both methods gave practically identical results. The multi-threaded pull-based method took 1,284.05 seconds, while the single threaded push-based method took 1,553.90 seconds. The top two tiles from the completed sets are presented in Figures 12.1 and 12.2.

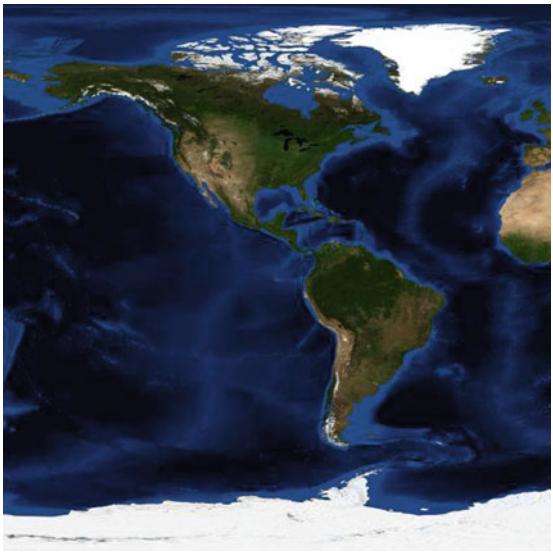

Fig. 12.1 Tile (0,0) from Blue Marble.

Fig. 12.2 Tile (0,1) from Blue Marble.

Listing 12.1 Example code for reading raw Blue Marble imagery.

```
public class RawImageReader {

    private String filename;
    int imageWidth;
    int imageHeight;
    private RandomAccessFile raf;
    private long bytesPerRow;
    public BoundingBox imageBounds;

    public RawImageReader(String filename, int width, int height, BoundingBox
         imageBounds) {
        this.filename = filename;
        try {
            this.raf = new RandomAccessFile(filename, "r");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        this.imageWidth = width;
        this.imageHeight = height;
        this.imageBounds = imageBounds;
        this.bytesPerRow = width * 3;
    }

    public synchronized BufferedImage getSubImage(int startx, int starty, int
         width, int height) {
        try {
            //create an empty image
            BufferedImage bi = new BufferedImage(width, height, BufferedImage.
                 TYPE_INT_RGB);
            //image is stored in row-major order
            for (int j = 0; j < height; j++) {
                //determine start position of the row to be read
                long startPosition = (starty + j) * bytesPerRow + startx * 3;
                //seek to the portion of the row that we need
                this.raf.seek(startPosition);

                for (int i = 0; i < width; i++) {
                    int r = this.raf.read();
                    int g = this.raf.read();
                    int b = this.raf.read();
                    //combine the rgb byte values into a single int value
                    int rgb = 0xff000000 | r << 16 | g << 8 | b;
                    //set the image pixel to the combined rgb value
                    bi.setRGB(i, j, rgb);
                }
            }
            return bi;
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;

    }


    public void close() {
        try {
            this.raf.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Listing 12.2 Cached clustered tile I/O.

```
1  public class CachedClusteredTileStream {
2
3      static final long magicNumber = 0x772211ee;
4      private String location;
5      private String setname;
6      private int numlevels;
7      private int breakpoint;
8      HashMap < String ,
9      BufferedImage > writeCache = new HashMap < String ,
10     BufferedImage > ();
11     HashMap < String ,
12     RandomAccessFile > openFileCache = new HashMap < String ,
13     RandomAccessFile > ();
14
15     public CachedClusteredTileStream(String location, String setname, int
           numlevels, int breakpoint) {
16         this.location = location;
17         this.setname = setname;
18         this.numlevels = numlevels;
19         this.breakpoint = breakpoint;
20     }
21
22     public void writeTile(long row, long column, int level, BufferedImage image
           ) {
23         String key = row + ":" + column + ":" + level;
24         writeCache.put(key, image);
25     }
26
27     public BufferedImage readTile(long row, long column, int level) {
28         String key = row + ":" + column + ":" + level;
29         if (writeCache.containsKey(key)) {
30             return writeCache.get(key);
31         } else {
32             ByteArrayInputStream bais = new ByteArrayInputStream(
33             readTileFromDisk(row, column, level));
34             BufferedImage bi = null;
35             try {
36                 bi = ImageIO.read(bais);
37             } catch (IOException e) {
38                 e.printStackTrace();
39             }
40
41             return bi;
42         }
43     }
44
45     public void writeTileFromCache(long row, long column, int level, byte[]
           imagedata) {
46
47         //first determine the cluster that will hold the data
48         ClusterAddress ca = getClusterForTileAddress(row, column, level);
49         String clusterFile = getClusterFileForAddress(ca);
50         if (clusterFile == null) {
51             return;
52         }
53         File f = new File(clusterFile);
54
55         //if the file doesn't exist, set up an empty cluster file
56         if (!f.exists()) {
57             createNewClusterFile(f, ca.endLevel - ca.startLevel + 1);
58         }
59         try {
60
61             RandomAccessFile raf = getOpenFileFromCache(f);
62
63             //write the tile and info at the end of the tile file
```

```
64              long tilePosition = raf.length();
65              raf.seek(tilePosition);
66              raf.writeLong(magicNumber);
67              raf.writeLong(magicNumber);
68              raf.writeLong(column);
69              raf.writeLong(row);
70              raf.writeInt(imagedata.length);
71              raf.write(imagedata);
72
73              //determine the position in the index of the tile address
74              long indexPosition = getIndexPosition(row, column, level);
75              raf.seek(indexPosition);
76
77              //write the tile position and size in the index
78              raf.writeLong(tilePosition);
79              raf.writeInt(imagedata.length);
80
81          } catch (Exception e) {
82              e.printStackTrace();
83          }
84      }
85
86      public byte[] readTileFromDisk(long row, long column, int level) {
87          //first determine the cluster that will hold the data
88          ClusterAddress ca = getClusterForTileAddress(row, column, level);
89          String clusterFile = getClusterFileForAddress(ca);
90          if (clusterFile == null) {
91              return null;
92          }
93          File f = new File(clusterFile);
94
95          try {
96              RandomAccessFile raf = getOpenFileFromCache(f);
97
98              //determine the position in the index of the tile address
99              long indexPosition = getIndexPosition(row, column, level);
100             raf.seek(indexPosition);
101             long tilePosition = raf.readLong();
102             int tileSize = raf.readInt();
103             if (tilePosition == -1L) {
104                 //tile is not in the cluster
105                 raf.close();
106                 return null;
107             }
108             byte[] imageData = new byte[tileSize];
109             //offset tile position for header information
110             long tilePositionOffset = tilePosition + 8 + 8 + 8 + 8 + 4;
111             raf.seek(tilePositionOffset);
112             raf.readFully(imageData);
113
114             return imageData;
115         } catch (Exception e) {
116             e.printStackTrace();
117         }
118         return null;
119     }
120
121     private long getIndexPosition(long row, long column, int level) {
122         ClusterAddress ca = this.getClusterForTileAddress(row, column, level);
123         //compute the local address, that's the relative address of the tile in
                the cluster
124         int localLevel = level - ca.startLevel;
125         long localRow = (long)(row - (Math.pow(2, localLevel) * ca.row));
126         long localColumn = (long)(column - (Math.pow(2, localLevel) * ca.column
                ));
127         int numColumnsAtLocalLevel = (int) Math.pow(2, localLevel);
```

```
128         long indexPosition = this.getCumulativeNumTiles(localLevel - 1) +
                localRow * numColumnsAtLocalLevel + localColumn;
129         //multiply index position times byte size of a tile address
130         indexPosition = indexPosition * (8 + 4);
131         return indexPosition;
132     }
133
134     public ClusterAddress getClusterForTileAddress(long row, long column, int
            level) {
135         if (level > this.numlevels) {
136             //error, level is outside of ok range
137             return null;
138         }
139         int targetLevel = 0;
140         int endLevel = 0;
141         if (level < breakpoint) {
142             //tile goes in one of top two clusters
143             targetLevel = 1;
144             endLevel = breakpoint - 1;
145         } else {
146             //tile goes in bottom cluster
147             targetLevel = this.breakpoint;
148             endLevel = this.numlevels;
149         }
150         //compute the difference between the target cluster level and the tile
                level
151         int powerDiff = level - targetLevel;
152         //level factor is the number of tiles at level "level" for a cluster
                that starts at "target level"
153         double levelFactor = Math.pow(2, powerDiff);
154         // divide the row and column by the level factor to get the row and
                column address of the cluster we are using
155         long clusterRow = (int) Math.floor(row / levelFactor);
156         long clusterColumn = (int) Math.floor(column / levelFactor);
157         ClusterAddress ca = new ClusterAddress(clusterRow, clusterColumn,
                targetLevel, endLevel);
158         return ca;
159     }
160
161     String getClusterFileForAddress(ClusterAddress ca) {
162         String filename = this.location + "/" + this.setname + "-" + ca.
                startLevel + "-" + ca.row + "-" + ca.column + ".cluster";
163         return filename;
164     }
165
166     //this methods create an empty file and fills the index with null values
167     void createNewClusterFile(File f, int numlevels) {
168         RandomAccessFile raf;
169         try {
170             raf = getOpenFileFromCache(f);
171             raf.seek(0);
172             long tiles = this.getCumulativeNumTiles(numlevels);
173             for (long i = 0; i < tiles; i++) {
174                 raf.writeLong(-1L); //NULL position of tile
175                 raf.writeLong(-1L); //NULL size of tile
176             }
177         } catch (Exception e) {
178             e.printStackTrace();
179         }
180     }
181
182     public int getCumulativeNumTiles(int finallevel) {
183         int count = 0;
184         for (int i = 1; i <= finallevel; i++) {
185             count += (int)(Math.pow(2, 2 * i - 2));
186         }
187         return count;
```

```
188     }
189
190     public RandomAccessFile getOpenFileFromCache(File f) {
191         String key = f.getAbsolutePath();
192         if (openFileCache.containsKey(key)) {
193             return openFileCache.get(key);
194         } else {
195             try {
196                 RandomAccessFile raf = new RandomAccessFile(f, "rw");
197                 openFileCache.put(key, raf);
198                 return raf;
199             } catch (FileNotFoundException e) {
200                 e.printStackTrace();
201             }
202         }
203         return null;
204     }
205
206     public void close() {
207         //iterate over tiles in the cache and write them to disk
208         Set < String > keys = writeCache.keySet();
209         for (String s: keys) {
210             String[] data = s.split(":");
211             long row = Long.parseLong(data[0]);
212             long column = Long.parseLong(data[1]);
213             int level = Integer.parseInt(data[2]);
214             BufferedImage image = writeCache.get(s);
215             ByteArrayOutputStream baos = new ByteArrayOutputStream();
216             try {
217                 ImageIO.write(image, "jpg", baos);
218             } catch (IOException e) {
219                 e.printStackTrace();
220             }
221             byte[] imagedata = baos.toByteArray();
222             writeTileFromCache(row, column, level, imagedata);
223         }
224         Set < String > openFiles = openFileCache.keySet();
225         for (String f: openFiles) {
226             RandomAccessFile raf = openFileCache.get(f);
227             try {
228                 raf.close();
229             } catch (IOException e) {
230                 e.printStackTrace();
231             }
232         }
233     }
234 }
```

Listing 12.3 Thread Safe Tile Range Iterator.

```
1  public class TileRangeIterator {
2
3      long curcol,
4      currow,
5      maxrow,
6      maxcol,
7      mincol,
8      minrow;
9      int level;
10
11     public TileRangeIterator(long minrow, long maxrow, long mincol, long maxcol
           , int level) {
12         this.minrow = minrow;
13         this.maxrow = maxrow;
14         this.mincol = mincol;
15         this.maxcol = maxcol;
```

```
16
17          this.curcol = mincol;
18          this.currow = minrow;
19          this.level = level;
20
21      }
22
23      public boolean hasMoreTiles() {
24          if ((this.currow <= this.maxrow)) {
25              return true;
26          }
27          return false;
28      }
29
30      public synchronized TileAddress getNextTileID() {
31          TileAddress address = new TileAddress(this.currow, this.curcol, this.
                level);
32          this.curcol++;
33          if (this.curcol > this.maxcol) {
34              this.currow++;
35              this.curcol = this.mincol;
36          }
37          return address;
38
39      }
40  }
```

Listing 12.4 Pull-based tile creation for Blue Marble.

```
1   public class PullTileCreation {
2
3       static int TILE_SIZE = 512;
4
5       public static void main(String[] args) {
6
7           BoundingBox imageBounds = new BoundingBox(-180, -90, 180, 90);
8           int imageWidth = 86400;
9           int imageHeight = 43200;
10
11          int baselevel = 8;
12          int numthreads = 8;
13          CachedClusteredTileStream cts = new CachedClusteredTileStream("folder",
                "bluemarble",baselevel,baselevel + 1);
14
15          RawImageReader imageReader = new RawImageReader("world.topo.bathy
                .200407.3x86400x43200.bin",imageWidth, imageHeight,imageBounds);
16
17          //initilize values for base level
18
19          long startRow = 0;
20          long startColumn = 0;
21          long endRow = TileStandards.zoomRows[baselevel] - 1;
22          long endColumn = TileStandards.zoomColumns[baselevel] - 1;
23
24          //build tiles for base level
25
26          //initilize the tile range iterator
27          TileRangeIterator tri = new TileRangeIterator(startRow, endRow,
                startColumn, endColumn, baselevel);
28
29          //create and start the tiling threads
30          Thread[] threads = new Thread[numthreads];
31          for (int i = 0; i < threads.length; i++) {
32              threads[i] = new PullTilerThread(tri, cts, imageReader);
33              threads[i].start();
34          }
```

```
35          //wait for the threads to finish
36          for (int i = 0; i < threads.length; i++) {
37              try {
38                  threads[i].join();
39              } catch (InterruptedException e) {
40                  e.printStackTrace();
41              }
42          }
43
44          //iterate over the remaining levels
45          for (int level = baselevel - 1; level >= 1; level--) {
46              int ratio = (int) Math.pow(2, baselevel - level);
47              long curMinCol = (long) Math.floor(startColumn / ratio);
48              long curMaxCol = (long) Math.floor(endColumn / ratio);
49              long curMinRow = (long) Math.floor(startRow / ratio);
50              long curMaxRow = (long) Math.floor(endRow / ratio);
51              //Iterate over the tile set coordinates.
52              for (long c = curMinCol; c <= curMaxCol; c++) {
53                  for (long r = curMinRow; r <= curMaxRow; r++) {
54                                  //For each tile, do the following:
55                      TileAddress address = new TileAddress(r, c, level);
56                      //Determine the FOUR tiles from the higher level that
                            contribute to the current tile.
57                      TileAddress tile00 = new TileAddress(r * 2, c * 2, level +
                            1);
58                      TileAddress tile01 = new TileAddress(r * 2, c * 2 + 1,
                            level + 1);
59                      TileAddress tile10 = new TileAddress(r * 2 + 1, c * 2,
                            level + 1);
60                      TileAddress tile11 = new TileAddress(r * 2 + 1, c * 2 + 1,
61                                                           level + 1);
62                      //Retrieve the four tile images, or as many as exist.
63
64                      BufferedImage image00 = cts.readTile(tile00.row, tile00.
                            column,
65                                                           tile00.level);
66                      BufferedImage image01 = cts.readTile(tile01.row, tile01.
                            column,
67                                                           tile01.level);;
68                      BufferedImage image10 = cts.readTile(tile10.row, tile10.
                            column,
69                                                           tile10.level);
70                      BufferedImage image11 = cts.readTile(tile11.row, tile11.
                            column,
71                                                           tile11.level);
72                      //Combine the four tile images into a single, scaled-down
                            image.
73                      BufferedImage tileImage = new BufferedImage(
74                                                                  TILE_SIZE,
75                                                                  TILE_SIZE,
76                                                                  BufferedImage.
                                                                        TYPE_INT_RGB
                                                                        );
77                      Graphics2D g = (Graphics2D) tileImage.getGraphics();
78                      g.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
79                                         RenderingHints.
                                               VALUE_INTERPOLATION_BILINEAR);
80                      boolean hadImage = false;
81                      if ((image00 != null)) {
82                          g.drawImage(image00, 0, Constants.TILE_SIZE_HALF,
83                                      Constants.TILE_SIZE_HALF, Constants.
                                            TILE_SIZE,
84                                      0, 0, Constants.TILE_SIZE, Constants.
                                            TILE_SIZE,
85                                      null);
86                          hadImage = true;
87                      }
```

```
88                      if ((image01 != null)) {
89                          g.drawImage(image01, Constants.TILE_SIZE_HALF,
90                                  Constants.TILE_SIZE_HALF, Constants.
                                        TILE_SIZE,
91                                  Constants.TILE_SIZE, 0, 0, Constants.
                                        TILE_SIZE,
92                                  Constants.TILE_SIZE, null);
93                          hadImage = true;
94                      }
95                      if ((image10 != null)) {
96                          g.drawImage(image10, 0, 0, Constants.TILE_SIZE_HALF,
97                                  Constants.TILE_SIZE_HALF, 0, 0,
98                                  Constants.TILE_SIZE, Constants.TILE_SIZE,
                                        null);
99                          hadImage = true;
100                     }
101                     if ((image11 != null)) {
102                         g.drawImage(image11, Constants.TILE_SIZE_HALF, 0,
103                                 Constants.TILE_SIZE, Constants.
                                        TILE_SIZE_HALF,
104                                 0, 0, Constants.TILE_SIZE, Constants.
                                        TILE_SIZE,
105                                 null);
106                         hadImage = true;
107                     }
108                     //save the completed tiled image to the tile storage
                            mechanism.
109                     if (hadImage) {
110                         cts.writeTile(address.row, address.column, address.
                                level,
111                                         tileImage);
112
113                     }
114                 }
115             }
116         }
117         cts.close();
118     }
119
120     public static Rectangle convertCoordinates(BoundingBox imageBounds,
            BoundingBox subImageBounds,int imageWidth, int imageHeight) {
121
122         int x = (int) Math.round((subImageBounds.minx - imageBounds.minx) / (
                imageBounds.maxx - imageBounds.minx) * imageWidth);
123         int y = imageHeight - (int) Math.round((subImageBounds.maxy -
                imageBounds.miny) / (imageBounds.maxy - imageBounds.miny) *
                imageHeight);
124         int width = (int) Math.round((subImageBounds.maxx - subImageBounds.minx
                ) / (imageBounds.maxx - imageBounds.minx) * imageWidth);
125         int height = (int) Math.round((subImageBounds.maxy - subImageBounds.
                miny) / (imageBounds.maxy - imageBounds.miny) * imageHeight);
126         Rectangle r = new Rectangle(x, y, width, height);
127         return r;
128     }
129
130     public static void drawImageToImage(BufferedImage source, BoundingBox
            source_bb,
131                                         BufferedImage target, BoundingBox
                                                target_bb) {
132         double xd = target_bb.maxx - target_bb.minx;
133         double yd = target_bb.maxy - target_bb.miny;
134         double wd = (double) target.getWidth();
135         double hd = (double) target.getHeight();
136         double targdpx = xd / wd;
137         double targdpy = yd / hd;
138         double srcdpx = (source_bb.maxx - source_bb.minx) / source.getWidth();
139         double srcdpy = (source_bb.maxy - source_bb.miny) / source.getHeight();
```

```
140         int tx = (int) Math.round(((source_bb.minx - target_bb.minx) / targdpx)
                );
141         int ty = target.getHeight() - (int) Math.round(((source_bb.maxy -
                target_bb.miny) / yd) * hd);
142
143         int tw = (int) Math.ceil(((srcdpx / targdpx) * source.getWidth()));
144         int th = (int) Math.ceil(((srcdpy / targdpy) * source.getHeight()));
145         Graphics2D target_graphics = (Graphics2D) target.getGraphics();
146
147         //use one of these three statements to set the interpolation method to
                be used
148         target_graphics.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
                RenderingHints.VALUE_INTERPOLATION_BILINEAR);
149
150         target_graphics.drawImage(source, tx, ty, tw, th, null);
151     }
152 }
153
154
155 public class PullTilerThread extends Thread {
156
157     private TileRangeIterator tri;
158     private CachedClusteredTileStream cts;
159     private RawImageReader imageReader;
160
161     public PullTilerThread(TileRangeIterator tri, CachedClusteredTileStream cts
            , RawImageReader imageReader) {
162         this.tri = tri;
163         this.cts = cts;
164         this.imageReader = imageReader;
165     }
166
167     public void run() {
168         while (this.tri.hasMoreTiles()) {
169             TileAddress address = this.tri.getNextTileID();
170             //Compute the geographic bounds of the specific tile.
171             BoundingBox tileBounds = address.getBoundingBox();
172             //get the bounds of the sub-image
173             Rectangle rect = PullTileCreation.convertCoordinates(
174             imageReader.imageBounds, tileBounds, imageReader.imageWidth,
                    imageReader.imageHeight);
175             //extract the image data from the source image
176             BufferedImage subImage = imageReader.getSubImage(rect.x, rect.y,
                    rect.width, rect.height);
177             //create a new empty image
178             BufferedImage tileImage = new BufferedImage(PullTileCreation.
                    TILE_SIZE, PullTileCreation.TILE_SIZE, BufferedImage.
                    TYPE_INT_RGB);
179             //scale the source image to the new image
180             PullTileCreation.drawImageToImage(subImage, tileBounds, tileImage,
                    tileBounds);
181             if (tileImage != null) {
182                 //write the image to the cache
183                 cts.writeTile(address.row, address.column, address.level,
                        tileImage);
184
185             }
186         }
187     }
188 }
```

Listing 12.5 Push-based tile creation for Blue Marble.

```
1  public class PushTileCreation {
2
3      static int TILE_SIZE = 512;
4
5      public static void main(String[] args) {
6
7          int baseLevel = 8;
8          String folder = "folder";
9
10         //create source image records
11         SourceImage a1 = new SourceImage("A1.jpg", new BoundingBox(-180, 0,
               -90, 90), 21600, 21600);
12         SourceImage b1 = new SourceImage("B1.jpg", new BoundingBox(-90, 0, 0,
               90), 21600, 21600);
13         SourceImage c1 = new SourceImage("C1.jpg", new BoundingBox(0, 0, 90,
               90), 21600, 21600);
14         SourceImage d1 = new SourceImage("D1.jpg", new BoundingBox(90, 0, 180,
               90), 21600, 21600);
15         SourceImage a2 = new SourceImage("A2.jpg", new BoundingBox(-180, -90,
               -90, 0), 21600, 21600);
16         SourceImage b2 = new SourceImage("B2.jpg", new BoundingBox(-90, -90, 0,
                0), 21600, 21600);
17         SourceImage c2 = new SourceImage("C2.jpg", new BoundingBox(0, -90, 90,
               0), 21600, 21600);
18         SourceImage d2 = new SourceImage("D2.jpg", new BoundingBox(90, -90,
               180, 0), 21600, 21600);
19
20         SourceImage[] images = new SourceImage[] {
21             a1,
22             b1,
23             c1,
24             d1,
25             a2,
26             b2,
27             c2,
28             d2
29         };
30
31         //create output stream to store tiles
32         CachedClusteredTileStream cts = new CachedClusteredTileStream("folder2"
               , "bluemarble", baseLevel, baseLevel + 1);
33
34         //build base level
35         for (int i = 0; i < images.length; i++) {
36             SourceImage currentImage = images[i];
37             BoundingBox currentBounds = currentImage.bb;
38             //determine the tile bounds specific to each source image
39             long mincol = (long) Math.floor((currentBounds.minx + 180.0) /
                   (360.0 / Math.pow(2.0, (double) baseLevel)));
40             long maxcol = (long) Math.floor((currentBounds.maxx + 180.0) /
                   (360.0 / Math.pow(2.0, (double) baseLevel)));
41             long minrow = (long) Math.floor((currentBounds.miny + 90.0) /
                   (180.0 / Math.pow(
42             2.0, (double) baseLevel - 1)));
43             long maxrow = (long) Math.floor((currentBounds.maxy + 90.0) /
                   (180.0 / Math.pow(
44             2.0, (double) baseLevel - 1)));
45
46             //if the image bounds go beyond the allowed tile bounds, set them
                   to the proper range
47             if (maxrow >= TileStandards.zoomRows[baseLevel]) {
48                 maxrow = TileStandards.zoomRows[baseLevel] - 1;
49             }
50             if (maxcol >= TileStandards.zoomColumns[baseLevel]) {
51                 maxcol = TileStandards.zoomColumns[baseLevel] - 1;
52             }
```

```
53
54              //read the source image from disk
55              BufferedImage bi = null;
56              try {
57                  bi = ImageIO.read(new File(folder + "/" + currentImage.name));
58              } catch (IOException e) {
59                  e.printStackTrace();
60              }
61              //iterate over the current tile bounds and create the tiled images
62              for (long c = mincol; c <= maxcol; c++) {
63                  for (long r = minrow; r <= maxrow; r++) {
64                      TileAddress address = new TileAddress(r, c, baseLevel);
65                      BoundingBox tileBounds = address.getBoundingBox();
66                      //check the cache for a pre-existing tiled image,
67                      BufferedImage tileImage = cts.readTile(address.row, address
                            .column, address.level);
68                      if (tileImage == null) {
69                          //the image wasn't in the cache, so create a new one
70                          tileImage = new BufferedImage(TILE_SIZE, TILE_SIZE,
                                BufferedImage.TYPE_INT_ARGB);
71                          cts.writeTile(address.row, address.column, address.
                                level, tileImage);
72                      }
73
74                      drawImageToImage(bi, currentBounds, tileImage, tileBounds);
75
76                  }
77              }
78          }
79          //iterate over the remaining levels
80          for (int level = baseLevel - 1; level >= 1; level--) {
81              long curMinCol = 0;
82              long curMaxCol = TileStandards.zoomColumns[level] - 1;
83              long curMinRow = 0;
84              long curMaxRow = TileStandards.zoomRows[level] - 1;
85              //Iterate over the tile set coordinates.
86              for (long c = curMinCol; c <= curMaxCol; c++) {
87                  for (long r = curMinRow; r <= curMaxRow; r++) {
88                      //For each tile, do the following:
89                      TileAddress address = new TileAddress(r, c, level);
90                      //Determine the FOUR tiles from the higher level that
                            contribute to the current tile.
91                      TileAddress tile00 = new TileAddress(r * 2, c * 2, level +
                            1);
92                      TileAddress tile01 = new TileAddress(r * 2, c * 2 + 1,
                            level + 1);
93                      TileAddress tile10 = new TileAddress(r * 2 + 1, c * 2,
                            level + 1);
94                      TileAddress tile11 = new TileAddress(r * 2 + 1, c * 2 + 1,
                            level + 1);
95                      //Retrieve the four tile images, or as many as exist.
96                      BufferedImage image00 = cts.readTile(tile00.row, tile00.
                            column, tile00.level);
97                      BufferedImage image01 = cts.readTile(tile01.row, tile01.
                            column, tile01.level);;
98                      BufferedImage image10 = cts.readTile(tile10.row, tile10.
                            column, tile10.level);
99                      BufferedImage image11 = cts.readTile(tile11.row, tile11.
                            column, tile11.level);
100                     //Combine the four tile images into a single, leveld-down
                            image.
101                     BufferedImage tileImage = new BufferedImage(
102                     TILE_SIZE, TILE_SIZE, BufferedImage.TYPE_INT_RGB);
103                     Graphics2D g = (Graphics2D) tileImage.getGraphics();
104                     g.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
                            RenderingHints.VALUE_INTERPOLATION_BILINEAR);
105                     boolean hadImage = false;
```

```
106                     if ((image00 != null)) {
107                         g.drawImage(image00, 0, Constants.TILE_SIZE_HALF,
                                Constants.TILE_SIZE_HALF, Constants.TILE_SIZE, 0,
                                0, Constants.TILE_SIZE, Constants.TILE_SIZE, null)
                                ;
108                         hadImage = true;
109                     }
110                     if ((image01 != null)) {
111                         g.drawImage(image01, Constants.TILE_SIZE_HALF,
                                Constants.TILE_SIZE_HALF, Constants.TILE_SIZE,
                                Constants.TILE_SIZE, 0, 0, Constants.TILE_SIZE,
                                Constants.TILE_SIZE, null);
112                         hadImage = true;
113                     }
114                     if ((image10 != null)) {
115                         g.drawImage(image10, 0, 0, Constants.TILE_SIZE_HALF,
                                Constants.TILE_SIZE_HALF, 0, 0, Constants.
                                TILE_SIZE, Constants.TILE_SIZE, null);
116                         hadImage = true;
117                     }
118                     if ((image11 != null)) {
119                         g.drawImage(image11, Constants.TILE_SIZE_HALF, 0,
                                Constants.TILE_SIZE, Constants.TILE_SIZE_HALF, 0,
                                0, Constants.TILE_SIZE, Constants.TILE_SIZE, null)
                                ;
120                         hadImage = true;
121                     }
122                     //save the completed tiled image to the tile storage
                            mechanism.
123                     if (hadImage) {
124                         cts.writeTile(address.row, address.column, address.
                                level, tileImage);
125                     }
126                 }
127             }
128         }
129         cts.close();
130     }
```