Chapter 3
Tiled Mapping Clients

A tiled mapping client has the responsibility of composing individual map tiles into a unified map display. The map display may allow the user to move around and load in more data, or it may be a static map image whose area is pre-determined by the application. These clients are generally not difficult to create (one of the benefits of tiled mapping). A tiled map client must be able to perform the following tasks:

- Calculate which tiles are necessary to fill the map.
- Fetch the tiles.
- Stitch tiles together on the map.

These three functions are usually performed in sequence as a response to an event in the map client (such as the user moving the map).

3.1 Tile Calculation

The first task of a tiled map client is to calculate which tiles are necessary to fill the map view. The map view is defined by both the geographic area of the map as well as the pixel size of the map. To perform the tile calculations, a simple function is necessary that takes the map view as input and returns a list of tiles as output. Each tile is defined by a tile scale, row, and column.

The actual implementation of such a function is dependent on the way map scale is handled within the client. For a tile client, the simplest method is to allow only a discrete set of map scales. The allowed scales are identical to the set of zoom levels provided natively by the underlying map tiles. Alternatively, the tile client may allow map scales that are not natively available in the tiled map. Usually, such a client will support a continuous set of map scales. Continuous map scales are used in GIS clients because their primary purpose is to create and analyze a wide variety of geospatial data. To support this extensive functionality, users must be able to work on data at any map scale. Supporting continuous map scales means that any combination of geographic area and map size is allowed. As a consequence, the

Listing 3.1 Calculation of tile range for the map view.

```
numMapViewTilesX= mapWidthPixels / tileWidthPixels
numMapViewTilesY = mapHeightPixels / tileHeightPixels
```

function that calculates which tiles are needed is more complex than the function used for discrete map scales.

3.1.1 Discrete Map Scales

The case where clients support only discrete map scales is the simpler one, so that is the best place to begin. As discussed in the previous chapter, the map scale of tiled imagery is specified by zoom level, which is not the traditional distance ratio (e.g., 1:10000m) common on paper maps. Instead, zoom level is simply used to specify the sequence of map scales supported by the tiles. Assuming a standard power of two tile scheme as discussed in the previous chapter, the world will be split into 2^{level} columns and $2^{level-1}$ rows ($level \in \mathbb{N}$). As level increases, so too does map scale. In this case, each increase in level squares the number of pixels used to represent the entire Earth. A map client that uses discrete scales will allow the user to choose only the levels made available by the data.

To calculate which tiles to retrieve, the client need only know the current zoom level, the tile index of the origin of the map view, and the number of tiles required to fill the map view. Notice that the map client need not know the geographic area of the map view. When the map client allows only discrete zoom levels, the state of the map may be stored using only a tile-based coordinate system rather than geographic one. It is also important to note that the following tile calculations assume that the map size is an integer multiple of the tile size. This assumption will be explained further later in the chapter, but it is a result of using common user interface programming techniques.

The origin of the map view is the index of the minimum tile. The minimum tile is the lower left tile when using Cartesian tile coordinates or the upper left tile when using matrix/image (row-major) coordinates. The tile dimensions of the map view are the width and height represented in tiles, i.e., the map width (height) in pixels divided by the tile width (height) in pixels. Listing 3.1 demonstrates how to calculate tile dimensions by dividing the map view size in pixels by the tile size in pixel for each dimension. Listings 3.2 and 3.3 demonstrate the process of increasing and decreasing the zoom level of the map. Both a new tile origin and tile range must be calculated when changing the zoom level.

Full-featured map clients will usually have the functionality to convert a geographic coordinate into tile coordinates as shown in Equations (3.1) (3.2).

Listing 3.2 Calculations for increasing the zoom level. When zoom level is increased we allow truncation when dividing.

```
# calculate the new horizontal tile origin index
 tileCenterX = numCanvasTilesX / 2 + tileOriginX
 newTileCenterX = tileCenterX * 2
 tileOriginX = newTileCenterX - numCanvasTilesX / 2

 # calculate the new vertical tile origin index
 tileCenterY = numCanvasTilesY / 2 + tileOriginY
 newTileCenterY = tileCenterY * 2
 tileOriginY = newTileCenterY - numCanvasTilesY / 2

 # calculate the new tile dimensions for the level
 tileRangeX = tileRangeX * 2
 tileRangeY = tileRangeY * 2
```

Listing 3.3 Calculations for decreasing the zoom level. When the zoom level is reduced we must round instead of truncating when calculating the tile origin.

```
# calculate the new tile dimensions for the level
tileRangeX = tileRangeX / 2
tileRangeY = tileRangeY / 2

# calculate the new horizontal tile origin
tileCenterX = numCanvasTilesX / 2 + tileOriginX
newTileCenterX = tileCenterX / 2
tileOriginX = int(round((numCanvasTilesX + tileOriginX) / 2.0)) -
       numCanvasTilesX
if (tileOriginX < 0):
      tileOriginX = 0

# calculate the new vertical tile origin
tileOriginY = int(round((numCanvasTilesY + tileOriginY) / 2.0)) -
       numCanvasTilesY
if (tileOriginY < 0):
      tileOriginY = 0
```

$$c = \frac{2^i(\lambda + 180)}{360} \tag{3.1}$$

$$r = \frac{2^{i-1}(\phi + 90)}{180} \tag{3.2}$$

where:

$$c \equiv \text{horizontal tile coordinate}$$

$$r \equiv \text{vertical tile coordinate}$$

$$\lambda \equiv \text{longitude; } -180 \le \lambda \le 180$$

$$\phi \equiv \text{latitude; } -90 \le \phi \le 90$$

$$i \equiv \text{discrete map scale}$$

3.1.2 Continuous Map Scales

Calculating the tile list for a map client with continuous map scales is more difficult. For a continuous scale client, the map view must be defined by the geographic area of the view and size of the view in pixels. From this definition of map view, it will be necessary to determine which zoom level is best used to populate the view and which specific tiles at that level are in the geographic area.

First, the current map scale must be calculated. The current map scale will still not be represented as a traditional distance ratio since this distance ratio varies over the entire world. Instead, the resolution for the map view, in degrees per pixel, will be used to represent map scale. It should be noted that this works only when the degrees per pixel is constant over the entire Earth at a particular zoom level; for certain map projections, this is not true, and the scale should be represented using the native coordinates of the projection (Chapter 10 has further discussion of map projections).

The degrees per pixel, *DPP*, may be calculated using the geographic area and size of the map view as shown in Equation (3.3).

$$DPP = \frac{DPP_x + DPP_y}{2} \tag{3.3}$$

where:

$$\begin{aligned}
DPP_x &= \frac{\lambda_1 - \lambda_0}{W} \\
DPP_y &= \frac{\phi_1 - \lambda_0}{H} \\
\lambda_1 &\equiv \text{maximum longitude of map view} \\
\lambda_0 &\equiv \text{minimum longitude of map view} \\
\phi_1 &\equiv \text{maximum latitude of map view} \\
\phi_0 &\equiv \text{minimum latitude of map view} \\
W &\equiv \text{width of map view in pixels} \\
H &\equiv \text{height of map view in pixels}
\end{aligned}$$

Each zoom level also has a fixed resolution associated with it. The degrees per pixel for each zoom level may be calculated using the tile geographic bounds and tile size with the above formula.

Once the resolution of the current map view is calculated, the process of determining the best zoom level for tile data may begin. Determining which zoom level to use as the source of tiles has important ramifications on image quality and client performance. If too low a zoom level is chosen, then the image will have an inappropriately low resolution and look pixelated. However, if too high a zoom level is chosen, then the client will be required to fetch too many tiles. For each increase in zoom level, the client must fetch four times the number of tiles to create any given map image. Thus, finding the optimal zoom level for a particular map view is important to the overall performance and quality of the map client. Of course, the optimal zoom level for a given map view is the zoom level with the same image resolution.

In general, the map view will not share an image resolution with any zoom level. Normally, the map view resolution will lie between the resolutions of two zoom levels. Of these two, the zoom level with the higher resolution is the best since it will reduce artifacts due to image scaling. However, a 10% margin of error is used when comparing the map view resolution to the resolution of the lower zoom level. If the map view resolution is within 10% of the lower zoom level resolution then the lower zoom level is used. A 10% resolution reduction is not significant visually and will not impact the resulting map image, whereas the four-fold savings in tile requests will provide significant performance improvements for the map client.

Often, the zoom level identified by the above process may not have the tiles necessary to compose the map view. In this case, an additional search must be conducted for the optimal zoom level tile source. If the upper bounding zoom level is not available, then the lower bounding zoom level should be used, even if the resolution is not within the 10% margin of error. If neither is available, then the next closest zoom levels should be checked, starting with the next highest zoom level. At most, only the next two higher zoom levels should be used. Beyond that the I/O costs of using higher zoom levels are prohibitive and should be avoided. Figure 3.1 shows examples of the processes of calculating the zoom level to use for a map view.

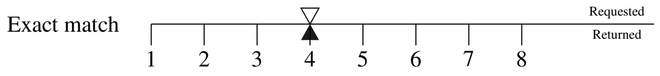
Once the appropriate zoom level is chosen as a tile source, the list of tiles covering the map view must be generated. The calculations for generating the tile list are similar to those used in the discrete map scale cases. First, the minimum tile must be calculated and then the tile range. However, in the continuous map scale case, the map view parameters are not integer multiples of the tile parameters. The minimum tile is calculated using the minimum point on the map view, which may be the lower-left or upper-left point depending on the tile coordinate system in use (we assume lower-left). Equations 3.4 and 3.5 are used to calculate the tile containing a geographic coordinate.

$$c = \lfloor (\lambda + 180.0) * \frac{360.0}{2^i} \rfloor \tag{3.4}$$

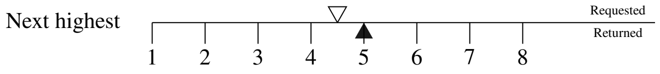
$$r = \lfloor (\phi + 90.0) * \frac{180.0}{2^{i-1}} \rfloor \tag{3.5}$$

where:

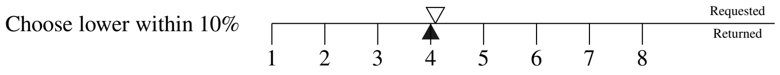
$$c \equiv \text{horizontal tile index}$$
$$r \equiv \text{vertical tile index}$$
$$\lambda \equiv \text{longitude of map view}$$
$$\phi \equiv \text{latitude of map view}$$
$$i \equiv \text{zoom level of map view}$$



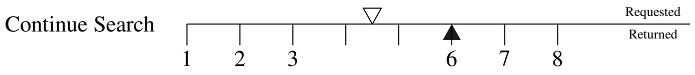
(a) The scale of the map view is exactly the same as an available tile zoom level.



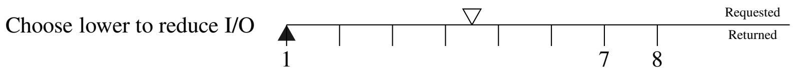
(b) The map view scale does not match an available zoom level, so we choose the next highest.



(c) The map view scale does not match an available zoom level, but it is less than 10% different than the next lowest tile zoom level. As a result we choose the lower tile zoom level.



(d) The map view scale lies in between two zoom levels that have no tiles available. In this case, we continue the search and choose the next highest zoom level.



(e) The map view scale lies in between two zoom levels that have no tiles available. Further surrounding zoom levels are also not available. In order to reduce the number of tiles to retrieve, we choose a lower zoom level, even though higher zoom levels are closer to the map scale.

Fig. 3.1 This figure shows examples of the process for choosing the appropriate zoom level.

The minimum and maximum coordinates of the map view are transformed into minimum and maximum tiles using these formulas. Once the list of tiles required to fill the map view is calculated, the client may proceed to the next major step of retrieving these tiles.

3.2 Tile Retrieval

The tile client must retrieve tiles from where they are stored to use them in the map view. Generally, the tiles are either stored locally or on a network. Sometimes tiles are stored using both mechanisms. Generally, it is a good practice to create an abstract interface for retrieving tiles, so the details of the implementation are separated from the rest of the map client functionality. That way, if the client must change or add an additional retrieval mechanism, the effect on the overall system is minimized.

3.2.1 *Local Tile Storage*

Local tile storage is the more complicated mechanism for tile retrieval. When tiles on disk are used, the map client must have internal knowledge of the tile storage scheme. In certain cases, the storage scheme is fairly simple. The single file-per-tile scheme is a prime example. Each layer, scale, row, and column combination identifies a single image file that can be easily referenced and exploited in the tile software. Also common is the database tile storage scheme. Here, the map client must be able to connect to the database and properly query for tiles. Database connections from software are trivial to implement these days making this method also relatively simple. More complicated are storage schemes where multiple tiles are stored in a single file. This type of storage system for tiles requires the map client to understand the organization of these files and most likely the indices used to find the tiles within them. Tile retrieval is more difficult in this case, but the performance benefits from such a storage scheme may outweigh the complications. Having a database dependency for a map client is not advisable, given the complexity of installing and managing databases. As discussed in chapter 7 on Tile Storage there are speed and space benefits to storing multiple tiles in a single file.

Generally, local storage of tile data should be limited in overall size. As the amount of tile data increases, so do the demands on the physical system supporting this data. It is usually not desirable to make a map client with large system requirements simply to support the accompanying data. Often, map clients will include one or two map layers with a limited base resolution. These are used as overview maps for the system and only provide a limited number of low zoom levels. Better map layers from external (i.e., network) sources are used for higher resolution data.

3.2.2 *Network Tile Retrieval*

Retrieval of map tiles from the network is a popular mechanism for map clients to get their data. For Web-based clients, it is a requirement. For desktop clients, it reduces the complexity and size of the software install. While network retrieval of tiles may be accomplished in a number of different ways, most commonly tiles are made available via Hypertext Transfer Protocol (HTTP; the protocol used for the Web). Specifically, each tile is retrieved by performing a GET via HTTP (one of five HTTP functions). By using HTTP GET, each tile is made available by a single URL. When accessing tiles over the network, the client need know nothing about the underlying storage mechanism for the tiles on the server. The server may store tiles in a database, as individual files, or in some custom file scheme, but this

Listing 3.4 Retrieving data from a URL in Python.

```
import urllib2
urlConnection = urllib2.urlopen('http://host.com/path/tile.jpg')
data = urlConnection.read()
```

is not reflected in the URL. Additionally, most programming languages provide libraries, which make retrieving data from a URL using an HTTP GET request a trivial process. Listing 3.4 contains an example of retrieving data from a URL in Python.

There are two common URL styles used in tile retrieval systems. The first encodes the tile parameters in the URL path. This method mirrors the path structure often used when storing tiles as individual files on the file system. An example path encoded URL is `http://host.com/tiles/bluemarble/3/5/2.jpg`. Here bluemarble is the layer name for the tiles, 3 is the zoom level, 5 is the tile column, and 2 is the tile row. The order of these parameters may change, especially the row and the column positions. The second URL style encodes the tile parameters in the URL parameters (the key-value pairs after the ? in the URL). The same map request encoded with URL parameters is `http://host.com/tiles?layer=bluemarble\&level=3\&col=5\&row=2`. Of course, both methods of encoding map requests in a URL have countless variations.

Layer management is another consideration for tile-based map clients to access network stored tiles. The simplest method of layer management is to simply hard-code the list of available layers inside the client. Hard-coding a layer list has the benefit of simplicity. No additional code is necessary to determine the list of layers available. However, hard-coding a layer list can be brittle. Whe the available layers changes, the map client must be updated to support the new layer list. Otherwise, the map client will create an error when a user tries to access a non-existent layer. In many cases the map client may be accessing data maintained by a third party. Without constant vigilance watching for changes in available layers, it is very likely that users will experience data problems.

A more robust alternative to hard-coded layers is to auto-detect the capabilities of each map tile service used by the client. The client may dynamically refresh the available services and layers so the user is always presented with a valid layer list. The result is fewer errors because of service changes or failures. Auto-detecting service capabilities is non-trivial when the tile service has a custom interface. Assuming the service provides a capabilities listing, the map client must have tailored code to parse the capabilities for available layers and supported zoom levels. The Web Mapping Tile Service (WMTS), as discussed in chapter 2, defines a standard capabilities listing format so clients may parse the capabilities of any compliant service.

The map client may choose to manage network errors and performance to improve the user experience. A complete loss of network access prevents the proper operation of a tiled map client, but limited functionality may be maintained by caching commonly used map tiles. The most commonly used tiled data are associated with low scales. Generally, users start with overviews of the entire Earth or a large area of the earth. These views require only a limited number of tiles since they use the lower zoom levels. Network performance may be improved by using tiles from a previous zoom level while a new one is loaded. When a user zooms the map view, the existing tiles may immediately be resized to fill the view. As new tiles come in, they may be placed above the zoomed lower level tiles. Once all tiles for the current zoom level are received, the zoomed tiles may be removed from the map.

3.3 Generating the Map View

After the map client retrieves tiles, it must use them to fill in the map view. The process of assembling tiles into a single map image varies depending on the technologies used to build the map client. However, the underlying process of generating a unified map view is essentially the same.

3.3.1 Discrete Scales Map View

Composing tiles into a single map view is simplest for discrete map scales. The process of determining which tiles to use was discussed above. Once the appropriate tiles are retrieved, they must be combined to form a single map view. The map view may simply be a static image or, more commonly, a portion of the user interface in map client software. Regardless, the algorithm to create the composite view is the same and relatively simple.

For the purpose of this section, we will assume the map client is a program that allows the user to interact with the map. The client must take the retrieved tiles and place them inside the map. The map will be the user interface container that holds the tiles. As stated earlier, the size of the map is an integer multiple of the tile size. The reasoning behind this assumption will be further discussed below, but for now we will hold it to be true. The user interface container used to hold the map varies depending on the programming language used to create the map client. In Java, the container would be a Panel or JPanel object (AWT and Swing respectively). In Python with the Tkinter user interface library, a Canvas object is used to hold images. Other popular programming languages have similar constructs. The following are a list of properties tht must be met by the container for it to function as a map:

- Hold multiple images.
- Absolutely position the images.
- Allow resizing of the container.

Assuming these conditions are met, the container will function appropriately as a map. It should be noted that an image has the above properties and may function as a map.

Placing the image tiles in the container is simple. The horizontal position of a tile is simply the tile index c multiplied by the tile size W. The vertical position must take into account the fact that the container most likely uses matrix coordinates; the upper-left corner is the origin. If the tile scheme uses Cartesian coordinates (the origin is at the lower-left) then a transformation must be made when placing the tiles in the container ($H_{contianer}$ is the container height, r is the vertical tile index, and H_{tile} is the tile height):

$$H_{container} - (r+1)H_{tile} \tag{3.6}$$

Once the coordinates for each tile are calculated, it may be placed in the container using absolute positioning.

The container holding the tile images is not the same as the map client application window. The application window, which we will call the viewport, will hold the tile container. The viewport is what the user actually sees. The viewport may be smaller or larger than the map container. Separating the size of the viewport from the size of the map container allows the map container to be fixed at an even multiple of the tile size, while the viewport size may vary arbitrarily. The map container will change size as the viewport is resized. The container should have width and height greater than or equal to the width and height of the viewport. At a minimum, round up the viewport width and height to the nearest multiple of tile size to calculate the map container dimensions. Often the map container will be sized to allow an unseen border of tiles one or two tiles deep. The border is used as a tile cache so that when the user moves the map, the tiles will appear on the map without requiring them to be fetched from the tile store. By prefetching unseen tiles, the apparent performance of a network tile store may be significantly improved.

3.3.2 Continuous Scales Map View

When the map client supports continuous scales, the tiled imagery may not be placed directly on the map. Instead, the tiles must be transformed to fit into the current map view. This task may be accomplished by performing the following three steps.

1. Stitch the tiles together into one large image.
2. Cut the large image to match the geographic area of the current map view.
3. Resize the cut image to the current pixel size of the map view.

These three steps are always required; however, some programming languages may provide user interface frameworks that simplify one or more of these steps. For this section we will assume the most basic of built-in functionality.

Stitching the tiles together may be accomplished by pasting them into one larger image. Stitching tiles together in an image is basically the same process as placing

Listing 3.5 Paste tiles into a larger image.

```python
# get the tile bounds from the geographic bounds
minTileX, minTileY, maxTileX, maxTileY = getTileBounds(bounds)

# retrieve the tile images from the datastore
tiles = fetchTiles(minTileX, minTileY, maxTileX, maxTileY)

# make PIL images
tileImages = makeImages(tiles)

# get the image mode (e.g. 'RGB') and size of the tile
mode = tileImage[0][0].mode
tileWidth, tileHeight = tileImage[0][0].size

# calculate the width and height of the large image to paste into
largeWidth = tileWidth * (maxTileX - minTileX + 1)
largeHeight = tileHeight * (maxTileY - minTileY + 1)

# make the new image with the correct mode, width, and height
largeImage = Image.new(mode, (largeWidth, largeHeight))

# loop through tiles and paste them into the large image
for row in xrange(maxTileY - minTileY + 1):
    for col in xrange(maxTileX - minTileX + 1):
        # calculate the location to paste the image (we use Cartesian
        # tile coordinates)
        x = col * tileWidth
        y = largeHeight - ((row+1) * tileHeight)
        # paste tiles into large image
        largeImage.paste(tileImage[row][col], (x,y))
```

them in the map container for a discrete scale client. First, make an empty image whose size is the combined width and height of all the retrieved tiles. Each tile should be pasted into the image according to its tile index. As with the discrete zoom level client, the tile indexing affects these calculations. If the tiles are Cartesian indexed (lower-left origin) then the vertical index must be transformed to align with the matrix indexing of images (upper-left origin). Listing 3.5 shows an example of stitching together images with Cartesian tile coordinates.

After the large image is created, it must be cut to match the geographic bounds of the map view. Each corner of the map view has a geographic coordinate. Each coordinate has a pixel location inside the stitched tile image. Those pixel coordinates are then used to cut the large image so its geographic bounds match those of the map view. Listing 3.6 contains code showing the process of cutting the map view.

The final step in creating an image to fill a continuous scale map view is to resize the cut image to have the same pixel size as the map view. First, the resolutions of the map view and the large image are calculated. The scaling factor for the large image is the ratio of the two resolutions. Usually, the resolution is represented as degrees per pixel. Listing 3.7 is an example of resizing the cut image to match the pixel size of the map view.

Listing 3.6 Cut the large image to match the geographic bounds of the map view.

```
# assume tileWidth = tileHeight
imageDPP = 360.0 / ((2 ** scale) * tileWidth)

# calculate the pixels for the rectangle to cut
leftPixel = int(round((mapViewMinX - imageMinX) / imageDPP))
lowerPixel = largeHeight - int(round((mapViewMinY - imageMinY) / imageDPP))
rightPixel = int(round((mapViewMaxX - imageMinX) / imageDPP))
upperPixel = largeHeight - int(round((mapViewMaxY - imageMinY) / imageDPP))

# cut out the rectangle
cutImage = largeImage.crop((leftPixel, upperPixel, rightPixel, lowerPixel)
```

Listing 3.7 Resizing the stitched together tiles to match the resolution of the map view.

```
# assume tileWidth = tileHeight
imageDPP = 360.0 / ((2 ** zoomLevel)*tileWidth)

# the view degrees per pixel may be calculated using the screen size
# and geographic bounds
viewDPP = getViewDPP()

scalingFactor = viewDPP / imageDPP
newWidth = int(cutWidth * scaling_factor)
newHeight = int(cutHeight * scaling_factor)
resizedImage = cutImage.resize((newWidth, newHeight))
```

3.4 Example Client

The code in Listing 3.8 contains a working example tile map client. This client is intended as an example to demonstrate some of the concepts discussed in this chapter. However, it is extremely simple and should not be considered an example of a user-ready map client. The example client uses discrete zoom levels and has limited movement controls. Movement is also limited to one tile at a time. The data source is a custom tile image server accessible using HTTP over the Internet. This client should work with a stock Python install along with the Python Image Library and libjpeg support. A screenshot of the example client is shown in Figure 3.2.

3.5 Survey of Tile Map Clients

A number of popular tile map clients exist and are heavily used by the geospatial community. Commonly used clients are either Web-based or desktop-based. Most Web-based clients allow only discrete zoom levels because it simplifies their design. Performing the image manipulation necessary to support continuous map scales would be difficult to support in a Web browser as well as overly costly. On the other hand, Web browsers support discrete zoom levels quite well because of their built-in

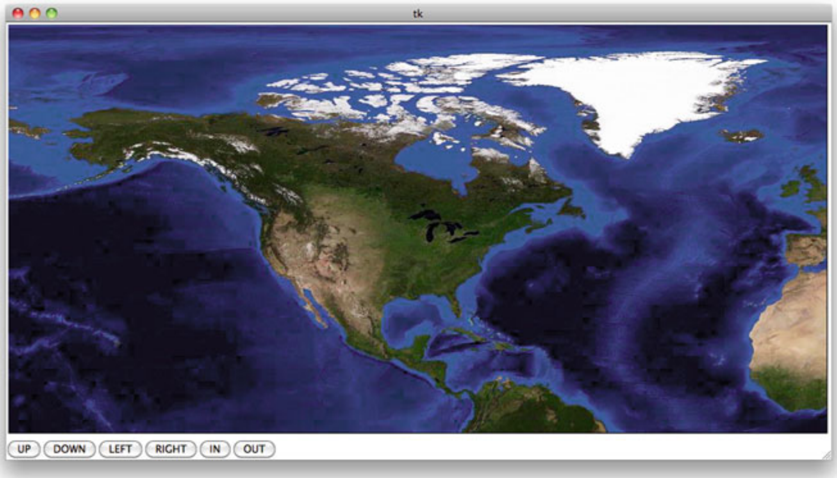


Fig. 3.2 Screenshot of the tile client example.

asynchronous design. When a discrete zoom level map client puts a tile on the map in a browser, it actually gives the image's network location to the browser and tells it to put the image on the map. The browser takes care of fetching the image asynchronously, so that while it is loading, the map client is still functional. Web-based clients are the most commonly used tile map clients. Commercial clients, such as Google Maps and Microsoft Bing Maps, are commonly used by a large non-expert audience to look at maps or get directions. These clients have developer interfaces so that custom data may be added along with the built-in data. However, as these clients are proprietary, they may not themselves be modified by a third party developer. The OpenLayers Web-based map client provides an open source alternative to the proprietary web clients. OpenLayers is a Javascript library that may be used with fewer restrictions than commercial clients.

Two popular desktop map clients are Google Earth and NASA World Wind. Google Earth has significant user penetration and allows developers to add arbitrary tile layers using KML network links. NASA provides World Wind as an open source program which supports full developer customization. Two versions of World Wind exist: a .NET version and a Java version. Both are open source. The .NET version is a full application, whereas the Java version is an SDK, intended for use in building custom map clients. Adding support for custom tile schemes is fairly simple in either.

Listing 3.8 Example Python map client which uses discrete zoom levels.

```python
import Tkinter
import Image, ImageTk   # requires the Python Imaging Library be installed
import urllib2
import cStringIO


class SimpleNetworkTileSource:
    '''
    This is a simple tile data source class to abstract the tile retrieval process from the rest of the client. Any class which implements this interface (i.e. has the same getTile method and member variables) may replace this source in the client.
    '''
    def __init__(self):
        self.layerNames = ['bluemarble']
        self.tileWidth = 512
        self.tileHeight = 512
        self.minLevel = 1
        self.maxLevel = 8

    def getTile(self, layerName, zoomLevel, colIndex, rowIndex):
        url = 'http://dmap.nrlssc.navy.mil/tiledb/layerserver?REQ=getimage' + '&layer=' + layerName + '&scale=' + \
            str(zoomLevel) + '&row=' + str(rowIndex) + '&col=' + str(colIndex)
        f = urllib2.urlopen(url)
        imageData = f.read()
        return imageData


class MapClient:
    '''
    MapClient is an implementation of a simple discrete zoom level map client.
    '''
    def __init__(self, parent, tileSource):
        # Starting zoom level
        self.level = 2

        # Starting tile origin
        self.tileOriginX = 0
        self.tileOriginY = 0

        # The number of tiles in each dimension for this level.
        self.tileRangeX = 2**self.level
        self.tileRangeY = 2**(self.level-1)

        # The size of the map view in tiles. We hardcode the size for this simple client
        self.numCanvasTilesX = 2
        self.numCanvasTilesY = 1

        # The tile source
        self.tileSource = tileSource

        # This dictionary is used to keep a reference to the tiles displayed in the UI so that they are not garbage collected.
        self.tileImages = {}

        # set up the user interface
        self.frame = Tkinter.Frame(parent, width=self.tileSource.tileWidth * self.numCanvasTilesX, height=self.tileSource.tileHeight * self.numCanvasTilesY)
        self.frame.pack()

        self.canvas = Tkinter.Canvas(self.frame, width=self.getFrameSize()[0], height=self.getFrameSize()[1])
        self.upButton = Tkinter.Button(self.frame, text="UP", command=self.up)
        self.downButton = Tkinter.Button(self.frame, text="DOWN", command=self.
            down)
        self.leftButton = Tkinter.Button(self.frame, text="LEFT", command=self.
            left)
        self.rightButton = Tkinter.Button(self.frame, text="RIGHT", command=
            self.right)
        self.inButton = Tkinter.Button(self.frame, text="IN", command=self.
            zoomIn)
        self.outButton = Tkinter.Button(self.frame, text="OUT", command=self.
            zoomOut)

        self.canvas.pack(side=Tkinter.TOP, fill=Tkinter.BOTH, expand=Tkinter.
            YES)
        self.canvas.create_rectangle(0,0,self.getMapSize()[0], self.getMapSize
            ()[1], fill='black')

        self.upButton.pack(side=Tkinter.LEFT)
        self.downButton.pack(side=Tkinter.LEFT)
        self.leftButton.pack(side=Tkinter.LEFT)
        self.rightButton.pack(side=Tkinter.LEFT)
        self.inButton.pack(side=Tkinter.LEFT)
        self.outButton.pack(side=Tkinter.LEFT)

        # load all the tiles onto the map for the first time
        self.loadTiles()

    def getFrameSize(self):
        return (int(self.frame.cget('width')),int(self.frame.cget('height')))

    def getMapSize(self):
        return (int(self.canvas.cget('width')),int(self.canvas.cget('height')))

    # Below are the controls for moving the map
    # up, down, left, right, zoom in, and zoom out
    #
    # After each is called loadTiles() is called to
    # refresh the map display with new tiles

    def up(self):
        if ((self.tileOriginY + self.numCanvasTilesY) < self.tileRangeY):
            self.tileOriginY += 1
            self.loadTiles()

    def down(self):
        if (self.tileOriginY > 0):
            self.tileOriginY -= 1
            self.loadTiles()

    def left(self):
        if (self.tileOriginX  > 0):
            self.tileOriginX -=1
            self.loadTiles()

    def right(self):
        if ((self.tileOriginX + self.numCanvasTilesX) < self.tileRangeX):
            self.tileOriginX += 1
            self.loadTiles()

    def zoomIn(self):
        if (self.level < self.tileSource.maxLevel):
            self.level += 1

            # calculate the new horizontal tile origin index
            tileCenterX = self.numCanvasTilesX / 2 + self.tileOriginX
            newTileCenterX = tileCenterX * 2
            self.tileOriginX = newTileCenterX - self.numCanvasTilesX / 2
```

```python

            # calculate the new vertical tile origin index
            tileCenterY = self.numCanvasTilesY / 2 + self.tileOriginY
            newTileCenterY = tileCenterY * 2
            self.tileOriginY = newTileCenterY - self.numCanvasTilesY / 2

            # calculate the new tile dimensions for the level
            self.tileRangeX = self.tileRangeX * 2
            self.tileRangeY = self.tileRangeY * 2
            self.loadTiles()

    def zoomOut(self):
        if (self.level > 2 and self.level > self.tileSource.minLevel):
            self.level -= 1

            # calculate the new tile dimensions for the level
            self.tileRangeX = self.tileRangeX / 2
            self.tileRangeY = self.tileRangeY / 2

            # calculate the new horizontal tile origin
            tileCenterX = self.numCanvasTilesX / 2 + self.tileOriginX
            newTileCenterX = tileCenterX / 2
            self.tileOriginX = int(round((self.numCanvasTilesX + self.tileOriginX) / 2.0)) - self.numCanvasTilesX
            if (self.tileOriginX < 0):
                self.tileOriginX = 0

            # calculate the new vertical tile origin
            self.tileOriginY = int(round((self.numCanvasTilesY + self.tileOriginY) / 2.0)) - self.numCanvasTilesY
            if (self.tileOriginY < 0):
                self.tileOriginY = 0

            self.loadTiles()


    # calculate which tiles to load based on the tile origin and
    # the size of the map view (in numCanvasTiles)
    def calcTiles(self):
        tileList = []
        for y in xrange(self.tileOriginY, self.tileOriginY + self.numCanvasTilesY):
            for x in xrange(self.tileOriginX, self.tileOriginX + self.numCanvasTilesX):
                tileList.append((x,y))
        return tileList

    # Get the tiles from the tile source and add them to the map view.
    # This is method is inefficient. We should really only fetch tiles
    # not already on the map. Instead we just refetch everything.
    def loadTiles(self):
        tileList = self.calcTiles()
        self.tileImages.clear()
        for tileIndex in tileList:
            # This is where the tiles are actually fetched.
            # A better client would make this asynchronous or place it in another thread
            # so that the UI doesn't freeze whenever new tiles are loaded.
            data = cStringIO.StringIO(self.tileSource.getTile(self.tileSource.layerNames[0], self.level, tileIndex[0], tileIndex[1]))
            im = Image.open(data)
            tkimage = ImageTk.PhotoImage(im)
            x = (tileIndex[0]-self.tileOriginX) * self.tileSource.tileWidth
            y = self.getMapSize()[1] - ((tileIndex[1]-self.tileOriginY+1) * self.tileSource.tileHeight)
            self.tileImages[tileIndex] = tkimage
            self.canvas.create_image(x, y, anchor=Tkinter.NW, image=tkimage)
```

```

if __name__ == '__main__':
    root = Tkinter.Tk()
    map = MapClient(root, SimpleNetworkTileSource())
    root.mainloop()
```