HW #7

JAY LAURA

For this assignment we were required to open and visualize a shapefile. My implementation makes use of the GeoSpatial Data Abstraction Library's (GDAL) OGR Simple Feature Library. This library is written in C++ and wrapped in a number of high level languages, including Python. With OGR it is possible to read and a myriad of vector formats, access geometry attributes via the attribute table, and reproject using OSR, a spatial reference library. Finally, OGR provides simple functions to convert between many OGC formats including WKT, WKB, KML, KMZ, GML, and geoJSON.

For this implementation I utilize argparse to simplify argument parsing. This is a Python 2.7+ module that replaces the old OptParse module. I utilize argparse for most complex command-line tools as it provides a straightforward tool to automatically generate command line accessible help documentation and packs arguments into a easily passable `dict`. Argparse also handles complex argument chains with easy, allowing for more complex options to be deployed. This program utilizes argparse only to store the input data set, e.g. the shapefile.

Instead of having to read the header and binary within the different components of a shapefile, OGR wraps that functionality in the `ogr.Open` function. If one were to print the returned value it would be a SWIG shadow and memory address. This is because the underlying C++ libraries are wrapped in SWIG.

Once the program has a handle to the shapefile we access the first layer. It is assumed that these shapefiles have a single layer, with a single geometry type as they confirm to the ESRI standard. It is possible that a shapefile, generated with OGR, could have multiple layers with divergent geometry types. I have never seen this in practice, but the capability does exist. Access to the layer essentially provides access the shapefile metadata and attribute table. In contrast to the previous homework assignment where we had to dynamically compute the minimum and maximum values for each axis, it is possible to simply query the extent of the shapefile. From this, the program computes the conversion ratio from coordinate space to pixel space.

To allow for the code to be more robust, the program queries the first row of the shapefile and returns the geometry type code. I parse the type code because different geometries are composed of different numbers of elements. For example, a point is simply a pair of coordinates, while a polygon is a ring, composed of pairs of coordinates. To access the points within a polygon, one must first access the ring. This software does not support complex geometries, but could be easily extended using simply Python flow controls. For example, by querying the number of rings that compose a polygon, it is possible to work with complex shapes (donuts) and multi-geometries.

Once parsed, the program accesses each point and performs the necessary coordinate to pixel space conversions. Note that the data is scaled to 95% total size and offset 10 pixels from the origin. This is because the `Tk.Canvas` object casts all `float` numbers to `int`. This causes geometries near the boundaries to be drawn on the boundary, i.e. converted points with values less than 1 are set to 0. This loss of precision alters the output geometry by clipping the edges. While scaling is not ideal, it does provide a easily implemented work around.

After converting all points and packing them into lists or lists of lists, the geometries are drawn using `Tk`. Note that `Tk` does not provide a point object by default and we draw points using the `Oval` function.

TABLE 1. Geometry Types and Type Codes

Geometry Code	Geometry Name
1	Point
2	Polyline
3	Polygon