Developer's Guide:

KML Klipper

Version 0.1.0

6-8-16

Rowan University S.E.G.V.

Project Team:	Dr. Adrian Rusu
	Dr. John Robinson
	Dr. Anthony Breitzman
	Robert J. Seedorf
	William Clark
	Eliakah Kakou
	Nicholas Laposta
Project Manager:	Dr. Adrian Rusu
Customer:	ASRC MSE Federal

Abstract

This document is the guide for developers of KML Klipper This document will elaborate on the implementation, external resources, case-by-case usage, and general implementation of ‘first project.’

1) Overview

   This guide will explain the usage of the development of KML Klipper for an end user. Using use case scenarios for the explanation of the project, we will convey the overall state, efficaciousness, and general implementation, of the program.

   There exist (two) implementations of our complete program that are each capable of accomplishing the desired goals, as per the requirements. These are each tailored to the desires of the operator and their choice of method for interfacing. These are the UI and the Console.

   The information this document reflects most of the modules of the program, and include much of the comment documentation for each.

2) System Requirements

   a) Python 2.7
   b) Recommended at least 1GB of RAM
   c) Internet Access (In order to utilize Google Maps API, Google Earth)
   d) Software with the capability to open, for viewing, image files

3) Dependencies
   a) Tkinter - a python package for GUI manufacture
   b) Webbrowser - a python package for local web browser interaction
   c) PIL - ‘python image library’, used for image file manipulation
   d) Lxml - a python library used in xml accessibility
   e) Pykml - a python library used in kml file accessibility
   f) Urllib - ‘url library’ a python library used for string url file manipulation

4) Modules
   a) Geometrics

i) class LatLongPoint(float latitude, float longitude)
    (1) This class is used as a container for the values of a single geographical point of a latitude and a longitude.
    (2) It contains a number in the domain of (-90, 90) for latitude and another number in the domain of (-180, 180) for longitude.
    (3) void rewrap()
        (a) The rewrap function takes the longitude of the LatLongPoint and converts the numbers to be between -180 and +180 if the coordinates are outside of that range.
    (4) (float, float) getTup()
        (a) The getTup function retrieves the latitude and longitude pair from the object as a tuple of floats
    (5) String rewriteStr()
        (a) The rewriteStr function retrieves the latitude and longitude as a single string separated by a comma
    (6) [float, float] listed()
        (a) The listed function retrieves the latitude and longitude pair from the object as a list of floats
ii) class GeometricObject
    (1) This class is used as a superclass to the following 3 classes. It is a wrapper for a single XML element that maps to geographic coordinates.
    (2) When creating a new element for the KML file a GeometricObject will have to be created and added to the KmlFasade.
    (3) Any changes desired to be made to a KML file will be made to a GeometricObject.
    (4) void applyEdits()
        (a) The applyEdits function applies any edits that have been made to the GeometricObject back to the KML file.
    (5) String printCoordinates()
        (a) The printCoordinates function prints all of the contained LatLongPoints as a single String separated by a pipe '|'.
    (6) [Strings] coordinatesAsListStrings()

(a) The coordinatesAsListStrings function returns all of the contained LatLongPoints as a list of individual Strings. Each LatLongPoint will be a single String

iii) class Point
  (1) See Geometric Object. This class specifies rules for a point xml object.
  (2) Void applyEdits()
    (a) See geometric object. This calls the super applyEdits, and if the removal flag has been set for this point, the point will be removed from the file.
    (b) This removal is done from the Placemark containing the point.
  (3) String printCoordinates()
  (4) String coordinatesAsListStrings(self)

iv) Class LinearRing(element, tag, coordinates)
  (1) See Geometric Object. This class specifies rules for a linearring xml object.
  (2) Void applyEdits()

v) class LineString(GeometricObject):
  (1) See Geometric Object. This class specifies rules for a LineString xml object
  (2) Void applyEdits()

vi) class Polygon
  (1) See Geometric Object. This class specifies rules for a Polygon xml object. For reference, It only contains the coordinates of the outer ring. Optional inner rings are ignored as a hollow iceberg is still an iceberg.
  (2) Void applyEdits()

vii) class GeometricFactory
  (1) Factory object to produce geometric objects. Can be extended to do input checking and other such utility functions.
  (2) Void createLiteral(element, tag, coordinates)
    (a) Generates a new Geometric object based on what the extracted tag is. This is important as certain xml objects require different functions to properly update changes.

(b) The literal means that this method takes the values directly. The create method, in comparison, takes the top level element and finds the values itself.

(3) Void create(element)

(a) Given an element, which is expected to have geometric data, create a geometric object for that data.

(b) Geometric objects wrapped tags such as Point, Polygon, and LinearRing, which have coordinate data. This method take the xml tag that starts a set of data, and processes until it has the required information to make a new Geometric When this factory is given a multigeometry object, it does an iteration through the multigeometry element calling create on each sub element. Those elements are returned as a list.

(4) String elementPrint(element, bool=0)

(a) Quick method to print an lxml element. For quicker writing.

viii) KmlFasade

(1) Class KMLFasade(path)

(a) This object wraps an lxml object and makes it easy to worth with.

(b) This is designed for quick, useful functionality that ignores irrelevant carry over data.

(c) It provides function to return list of useful xml data, tools to apply changes to the xml file based on the objects it generates, and other features.

(2) Void rewrite(path=None)

(a) Writes the stored file object back to the original file or a provided path. Basically combines a few lxml methods to make this task quicker.

(3) Void removeGarbageTags()

(a) This method is a catch all for and garbage filtering operations. Currently it has one functionality, it takes every element stored in the garbage field and removes it from the Kml. Each element in the garbage list has no relevant data, removing it helps with the file length.

(4) Void pullPlacemarksAndGarbage()

(a) This method is used to append any xml tag with the placemark tag to a list and return it. As the most relevant data in a kml file appears in a placemark tag, this is a convenience method to prevent excess searching. During the integration, the method also looks to find all elements in the file that are irrelevant. Each element with it's tag in garbage data has no geometric data we care about. We can freely delete them later with the garbageFilter method.

(5) List placemarkToGeometrics()

(a) This method takes the list of placemarks it has generated (or generates them) and creates geometric objects to allow for easy of editing.

(6) fasadeUpdate()

(a) Runs the applyedit function on every geometric object contained in this object's geometrics list. If the addition folder has been generated, This method will also add that folder to the file.

(7) Void createAdditionsFolder()

(a) Initializes the kml folder any additional points will be written to. Simple kml container.

(8) createAdditionalGeometry(type, name='blank', coordin='0,0')

(a) This method allows for a user to create new point, linestrings, and linearrings. It will default the name and coordinates for the object if they aren't provided, but every call requires a type, one of the above three. This method appends the created kml to the additions folder which needs to be produced beforehand.

(9) yieldGeometrics()

(a) Returns the geometrics contained in the class. Not really needed, but this helps for implementations using the composite module. It uses yield instead of return so that it can be accessed in the same way of the composite, for x in yield.

b) Mercator

i) Class Mercator(x=0, y=0)

(1) Container for the location of a pixel on a Mercator map projection

ii) Class MercatorProjection()
    (1) Contains conversion functions to convert a GeoPoint to a GeoLatLng and back using a center pixel of a Mercator map projection.
    (2) Creates a pixel coordinate for the center point of the map that every pixel coordinate will be relative to as well as the ratio of longitude degrees/radians to pixels
    (3) MercatorPoint from_lat_lng_to_point(lat_lng, opt_point=None)
        (a) Converts a GeoLatLng object into a GeoPoint object
    (4) LatLonPoint from_point_to_lat_lng(self, point)
        (a) Converts a GeoPoint object into a GeoPoint object
    (5) Float degrees_to_radians(deg)
        (a) Converts degrees to radians
    (6) Float radians_to_degrees(rad)
        (a) Converts radians to degrees
    (7) (LatLonPoint) get_corners(center, zoom, map_width, map_height)
        (a) This function returns a dict of all of the corner points and the four cardinal direction limits of the image that will be downloaded from the Google Maps API

c) RestrictionEngine
    i) class RestrictionFactory(metric=0):
        (1) This method generates a new restriction. The factory makes sure that every restriction created works off the same measurement scale, metric or imperial.
        (2) CircleRadiusRestriction newCircleRadiusRestriction(center, distance)
            (a) See CircleRadiusRestriction, this method returns a new instance of that object.
        (3) SquareRestriction newSquareRestriction(center, distance)
            (a) See SquareRestriction, this method returns a new instance of that object.
        (4) WAClippingRestriction(newWAClipping(viewport))
            (a) see MercatorRestriction, this method returns a new instance of the object
    ii) Class Restriction

(1) An 'interface' (as interface as python gets) for new Restrictions. With it templating and the composite methods can be built into the restriction engines.
(2) Void restrict(geometrics)
    (a) The primary method for a restriction. The only method that *should* be called from outside the class. Should modify the geometrics it gets, (not remove them) as the implementation chooses to do so.
(3) (Float, Float) zoom(width)
    (a) Determines the proper zoom level and filter range for the given frame size
(4) Float haversine(start, end)
    (a) Uses the mathematical function of the same name to find the distance between two long lat coordinates.

iii) class WAClippingRestriction(viewport)
(1) Void restrict(geometrics)
    (a) This restriction iterates all through the provided list of geometrics. On each iteration, the coordinates in the geometry are checked for two things; being within the viewport of the Restriction and if the point is an entry point.
    (b) An entry point is required for WA clipping to work,simply a point that the prior point was not in the viewport.
    (c) After the iteration, any geometry that is partially in the viewport is clipped via the WeilerAtherton module.
    (d) Boolean pointWithinCorners(coordinates)
        (i) Returns true if the given coordinates are contained by this classes NW and SE lines. Helper method to restrict.

iv) class SquareRestriction(center, distance, metric=0)
(1) A restriction that flags all points that are not within distance x from a given center point to be removed.
(2) This is done in a square pattern by using two points, the NW corner and the SW corner.
(3) Void restrict(geometrics)
    (a) This method restricts based off of the NW and SE values this object contains.

(b) It looks at each point in a geometric and checks to see if it's contained by the lines drawn from NW and SE. If it's contained, we know that the point is in our frame of mind.

v) class SquareRestriction(center, distance, metric=0)

(1) A restriction that flags all points that are not within distance x from a given center point to be removed. This is done in a square pattern by using two points, the NW corner and the SW corner.

(2) Void restrict(geometrics)

(a) This method restricts based off of the NW and SE values this object contains.

(b) It looks at each point in a geometric and checks to see if it's contained by the lines drawn from NW and SE. If it's contained, we know that the point is in our frame of mind.

(3) Boolean pointWithinDistance(self, coordinates)

(a) Returns true if the given coordinates are contained by this classes NW and SE lines. Helper method to restrict.

vi) class CircleRadiusRestriction(center, distance, metric=0)

(1) A restriction that flags all points that are not within distance x from a given center point to be removed.

(2) Void restrict(geometrics)

(a) Looks at each geometric object in the list and, if it is not within distance of center, flags it for removal.

d) WeilerAtherton

i) Class WeilerClipping(center, debug=0)

(1) This class is used to store all arithmetic procedures necessary to establish a uniform problem set, solve, and produce the solution of polygon clipping as it pertains to this project.

(2) This involves LatLonPoint transformation, construction of iterable, ordered sets of said points, and corresponding geometric lines of those points, calculation of the points at which the determined lines will intersect in space, and the production of the desired goal: the result of clipping one polygon against another

(3) LatLongPoint getLineIntersect(pointA, pointB, pointC, pointD)
    (a) Returns the point at which the two lines, lines AB and CD, intersect,
    (b) NOTE even if this point of intersection lies outside of the space delimited by their endpoints. As a precondition, it should be checked if they do or do not intersect
(4) Float determinant(pointA, pointB, pointC, pointD)
    (a) Finds the value of the algebraic determinant of the two matrices [a, b] and [c, d]
    (b) Used for point of intersection calulation
(5) Boolean doLinesIntersect(pointA, pointB, pointC, pointD)
    (a) This method will return two if the two line segments AB and CD intersect at some point in space
(6) Integer findOrientation(pointA, pointB, pointC)
    (a) Returns orientation of three given points in coordinate space.
    (b) The assumption lines drawn between points A, B and B, C and C, A are used to define this 'orientation'
    (c) Through projection of the combinations of lines and extraneous points the slopes of the projections are found to be of an appropriate orientation
(7) LatLonPoint getClosestPoint(target, pointA, pointB):
    (a) Returns the point of only either A or B that is closer to point Target
    (b) Requires all points must be of the class LatLonPoint class
(8) [LatLonPoint] getClipped(P, Q, Ie)
    (a) Through application of the Weiler Atherton Clipping algorithm this method finds the list of ordered points that compose the polygon that is the result of said clipping
(9) [LatLonPoint] getP(subjectlines, viewportlines)
    (a) Returns the ascribed collection P, the list that stores all points, including the points of intersection, that compose the subject polygon, traversed in a ccw motion
(10) [LatLonPoint] getQ(subjectlines, viewportlines)

(a) Returns the ascribed collection Q, the list that stores all points, including the points of intersection, that compose the viewport polygon, traversed in a ccw motion

(11) Void unwrap(list)

(a) Statically modifies all the LatLonPoints of the parameter list by actively modifying the elements to conform to the space of a one directional map outstretched from the easternmost side of the anti-meridian.

(b) This method's purpose is essential; to return anti-meridian crossing lines to real continuous space. If the LatLonPoints are not converted into a format where they can be used in continuous space, the results of clipping will never be right

(12) Void rewrap(list)

(a) Statically modifies all the LatLonPoints of the parameter list by actively calling rewrap on every point in the parameterized list of lists, that represent the lines of the paired points

(13) [LatLonPoints] unFlattenList(list)

(a) This method returns the result of converting a flattened list of points, stored in a contiguous fashion, into the nested list of lists, where internal lists are the line-wise objects of the contagious points

(14) Void Clip(subjectlines, viewportlines)

(a) 'Run-me' method that applies all of the organized operations in the correct fashion

e) ImageMerge

i) Class Merger

(1) This class is basically the same method as mergemodeRGB. The add method is almost identical.

(2) The reason for this class is that it allows for urls to be added incrementally with ease. Given an instance where you want to merge urls to the same outfile based off the same base image, but don't have the urls all at once,this method allows for any number of add calls to merge images to the output.

(3) This comes with overhead, saving the image to the outfile location after each individual merge, but certain situations really need this kind of functionality.

(4) Void merge(new)

- (a) Given a new image, with layers of paths and markers, on top of a plain base image. It will then output the merged image to the outfile provided.
- (b) This method only works with images encoded in RGB color, which is expected of it's inputs. The method works by opening the tracked changes file, and opening the base and the new image.
- (c) Each pixel in the base is compared with the new image, if they're different enough the new pixel is placed over the original base image pixel that was contained in the tracked file.
- (d) This is done for every pixel. Debug information contains the ratio of different to not different pixels in percent form, as well as a display of the output after each layer merge.

(5) Void mergeAll(*images)

- (a) This merges a list of images, with layers of paths and markers, on top of a plain base image.
- (b) It will then output the merged image to the outfile provided. This method only works with images encoded in RGB color, which is expected of it's inputs. The method works by creating a new file for the changes, and opening the base and a layer.
- (c) Each pixel in the base is compared with the layer, if they're different enough the layer pixel is placed over the original base image pixel that was contained in the new file.
- (d) This is done for every pixel and every layer. Debug information contains the ratio of different to not different pixels in percent form, as well as a display of the output after each layer merge.

(6) Void convertAll(*image)

- (a) This method takes any number of images and converts the from P color mode to RGB mode.

(b) RGB is required by the merge methods above. The files are saved to a different file, same prefix, but .con being placed at the end.

(7) Void blkDiff(outfile="DifferenceFile.png")

(a) This method will change the color of any pixel that is different between a base image and another provided image.

(b) This works in the same way as mergeRGB, which does mean the inputs need to be RGB format. You can use this to visually see the detected differences between two images.

f) UrlBuilder

i) class UrlBuilder(Observable)

(1) The UrlBuilder Class should be created anew for each url the user wants to build. Using the methods contained, the user may create a valid google maps static api url. Input is not checked,

(2) And all inputs should be strings.

(3) Even integer values should be entered repr()'d.All inputs will be need to be converted to Long Lat, as all other files use Lat Long.

(4) All URL references are stored as strings, in a url format

(5) URL addparam(feature, value)

(a) Appends the given parameter to the static maps url. This applies to all urls in a split scenario.

(6) URL addparams(dict)

(a) Allows for a list of parameters to be added to the static map. This is applied to all urls in a split scenario.

(7) URL centerparams(center, zoom)

(a) This method adds two parameters to the url that are required for a consistent image.

(b) In order to merge images, these parameters are required to keep the image displaying the same area. In a split scenario, these are applied to each url. Multiple calls are unnecessary.

(8) URL viewportparam(viewports)

(a) Appends a viewport parameter. A viewport makes each point it is given visible on the map.

(b) If the parameters are provided as a list, when a coordinate is added and the length is pushed over the character limit, the url will be split.

(c) If the locations are provided as a solid string, the url will only be checked after everything is appended. (This could cause data loss.)

(9) URL addmarkers(styles, locations)

(a) Adds the locations listed as markers to the url. Each point will have the supplied style settings. If the parameters are provided as a list, when a coordinate is added and the length is pushed over the character limit, the url will be split.

(b) If the locations are provided as a solid string, the url will only be checked after everything is appended. (This could cause data loss.)

(10) URL addpath(self, styles, locations)

(a) Adds the path listed to the url. The lines drawn and area filled will have the style setting specified.

(b) If the url is split, filling with not work, and can possibly break the url. Filled areas are expected to within a single url by google maps static.

(c) If the parameters are provided as a list, when a coordinate is added and the length is pushed over the character limit, the url will be split.

(d) If the locations are provided as a solid string, the url will only be checked after everything is appended. (This could cause data loss.)

(11) URL download(path="""Inputs\Static Maps\Mass\{} {}.png""", prefix='image')

(a) Takes a parameter path and downloads the generated url to that path. Using the symbol {} {} twice will replace the first with the given prefix, and the second with a counter.

(b) This is the recommended way of using this path, because multiple urls may be downloaded.

(12) URL downloadBase(path='Inputs\Static Maps\\Mass\{} 0.png', prefix='image')

(a) Downloads only the base url. The generator method does not download that url (better for automation), so this method was a required addition.

(13) Integer countUrl(url)

(a) This method counts the characters in the given url. Of note, This does not use internal values, unlike most methods in the class.

(b) This method is necessary because '|'' characters count as 3 when actually read by static maps

(14) Integer retreiveURL(url, offset=0)

(a) This method is used to split urls that have hit the size limit. It uses side effects to great effect.

(b) This method actually checks the url (and an optional offset) against the limit, and if the number is greater than or equal to the limit, returns true.

(c) When it does return true, it adds the current url to the url list and provides a new url copied from the urlbase saved in the object.

(d) This is how all parameters are kept the same across splits, parameters update the base url and not the actual current url. (technically they do update both, but you get the idea.) The method returns false if the url is shorter than the limit.

(15) [URL] printUrls()

(a) This method prints all the urls contained in the object in a readable manner. As readable as lines of character length 2000~ can be. It labels the base url (used for image merging) and then lists all layer images.

g) PathFinder

(i) Class Main

(1)This class executes all of the modules in the appropriate order based on the user's input.

(ii)Class Map(start, end, size, list(optional))

(1)InitNode()

(a) Initializes the nodes by checking generating the coordinates of each node. It also uses the point checker to determine if a node is walkable(usable) or not.

(2)findPath()

(a)Uses the A* algorithm to determine if there is a path and what it is. It then returns a list of nodes which make up the path found.

(3)drawMap()

(a)This module prints a text representation if the space where the nodes are located on the map. '#' indicates unwalkable nodes, and '0' indicates walkable nodes.

(4)getDistance()

(a)This method takes two nodes and returns the distance between them in double using their coordinates.

(iii)Class Node

(1)This class represents a node, it holds a Point object, and whether is it a usable node or not.

(iv)Class Point

(1)This class represents a point on the map. It holds its latitude and longitude. It also corrects the latitude and longitude of the point if the coordinates entered are outside the boundaries of the map.

(v)Class PointChecker

(1) execute(lat, lng)

(a)This module runs a PHP script that checks google maps' pixels at a certain coordinate.

(2) isInside(polygon, point)

(a) This module checks if the point is inside the polygon, then returns true/false.

(3) Index.php

(a) This script takes a latitude and longitude parameter which allows it to check for a pixel color a that coordinate. It then returns a 1 if the pixel is black and 0 if it is white.

(b) This script along with the pointchecker module is in the htdocs in the xampp folder. It runs as if it were on a server.

(vi)Class KMLParser(fileName)

(1)polygonCoord()

(a)This module reads a kml file then returns the coordinates of the points making up the polygon(s) in the file.

(viii)Class KMLGenerator(nodes, start, end, path, poly(optional))

(1)createPolygon(points)

(a)This module creates a polygon tag using the list of points Inputted and returns a string.

(2)createPlacemark(point)

(a)This module creates a placemark tag using the point inputted

and returns a string.

(3)createPath(nodes)

(a)This module creates a path tag the list of nodes inputted and returns a string.

5) Reference

a) Please see the document, titled ‘how_to.txt’, for user directions (how to guide)
b) For further info on usage, please see the SUM (software user manual)

6) Glossary

a) KML - the xml file format used by google’s earth viewing software, the medium in which out file examinations work
b) GUI - Graphic User Interface
c) ccw - ‘counter clockwise’
d) dict - ‘dictionary’ a in to hashmaps for python
e) Lxml - python package for xml operation