Projecting NYC 3-D Building Model onto Urban Observatory Images (May 2017)

Victor Sette Gripp, Anastasia Shegay, Vishwajeet Shelar, Aaron D'Souza

I. Introduction

The Urban Observatory (UO) at the NYU Center for Urban Science and Progress (CUSP) collects terabytes of image data of New York City-scapes from various vantage points in Manhattan and Brooklyn in order to gain insights into patterns of energy consumption and pollution along with other urban phenomena. However, these image data are of limited analytical use without integrating additional information about the objects one can observe in an image, for instance their geographic location or zoning characteristics. Projecting the geospatial data contained in the three-dimensional (3-D) building model of NYC onto a UO image unlocks wealth of information about the buildings seen in the image, such as the building identification number (BIN) and height. Extracting the BINs would also allow us to merge the UO image data with additional NYC datasets for further analysis, for instance land use data from MapPLUTO as well as socioeconomic and demographic data from the U.S. Census.

The goal of this project is to produce a generalizable script that would project the 3-D building model data onto an image captured by the UO, using photogrammetry and ray tracing techniques. While the discipline of photogrammetry, the "science of obtaining reliable information about the properties of surfaces and objects without physical contact with the objects" [1] dates back to the invention of modern photography, in recent years photogrammetry benefited from advances in computing and proliferation of data. This project in particular makes use of high-performance distributed computing resources to parallelize the photogrammetry process. The expected outcome of this project is an image with associated geographic locations for each visible building.

II. Data and Methods

The main dataset we used for this project is the 3-D Building Model of NYC [1]. The model includes every building in NYC captured by an aerial survey conducted in 2014. The dataset is made publicly available by the NYC Department of Information Technology and Telecommunications (DoITT). The attributes include latitude, longitude, building identification number (BIN), and height. The size of the originally obtained dataset was 995 MB.

The second dataset is an optical image taken in 2017 by the CUSP UO instrumentation from the vantage point at One Bryant Park - Bank of America Tower (Figure 1). While we can easily discern certain landmark buildings--e.g. the Empire State and One World Trade Center--it is difficult to identify other buildings in the image, especially those further away. The image dimensions are 1,918 x 2,560 pixels; the image thus contains a total of 4,910,080 pixels. The size of the image is 8.9 MB.

While the total size of the originally acquired data was not large, further data processing and techniques applied in the project entailed several "big data" challenges, which we describe in detail in section III.

Figure 1. The vantage point is One Bryant Park facing southward toward downtown Manhattan.

A. Preprocessing

The input data for our analysis was contained in three files:

a. The two dimensional image file (png format)
b. The three dimensional city model (Multipatch and DGN formats)
c. The MapPLUTO shapefile (shp format)

While the image file and the shapefile can directly be loaded in a Jupyter notebook, the DGN and multipath files containing the 3-D building model required preprocessing. These files were converted into numpy files containing the longitude, latitude and elevation of every point with one-foot resolution. This was done in the following steps:

1. The DGN and multipatch data were first sliced to contain only the data of lower Manhattan within the view of the camera.
2. This clipped data was rasterized in ArcGIS with a one foot resolution with each pixel denoting the elevation of the pixel. This raster was then discretized into 99 “tiles” for computational purposes. The tiles which were behind the camera were removed, resulting in a total of 78 tiles.
3. The raster tiles were then converted into CSV files to access it on Python and then converted into numpy files to be used for analysis.

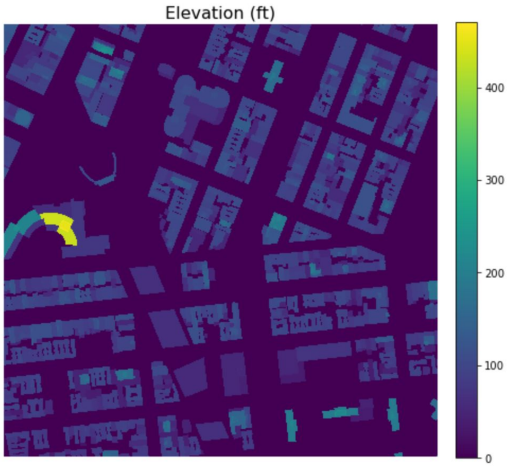

Figure 2. One of the resulting 78 tiles. The *x*-axis is the longitude, the *y*-axis is the latitude, color is the elevation.

B. Algorithms

This project adapted and optimized several existing photogrammetry algorithms previously used by the UO [3, 4].

Camera optimization. In order to obtain the camera parameters from an image, we implemented a camera optimization algorithm. This algorithm outputs the seven degrees of freedom of a camera such as camera location (x, y, z), pitch, yaw, roll, and focus. In order to solve a function to obtain of seven unknowns we need to solve seven equations simultaneously. For this purpose we chose ten unique “fiducial” points on the image and their corresponding locations were identified on both the image and the 3-D model. The fiducials on the image contained the x, y coordinate of the pixel and the 3-D model contained the x, y and z coordinates of the point on the model.

This camera optimization is done using a numerical method. Initially the parameters of the camera are chosen by an informed guess. These values are then passed to the collinearity function (Algorithm 2) along with the fiducials of the ten 3-D points to evaluate if the values are matching the pixel position of the seven fiducials of the image. The error value of the colinearity function and the image fiducials obtained were then minimized using a numerical method. This process was then iterated several times (with different initial guesses) to obtain a minimum value for the error term. The parameters corresponding to the function with the minimum error value are chosen as our parameters for the photogrammetry.

Algorithm 1. Photogrammetry to convert 3D into 2D.

Inputs: t= {$t_{(x1, y1, z1)}$, $t_{(x2, y2, z2)}$, $t_{(x3, y3, z3)}$, …} : an array of longitude, latitude and elevation
Parameters: params = {κ: roll, φ : phi, ω : omega, x_s : longitude, y_s : latitude, z_s : elevation, f : focus}
ImageDimension = {m : no of pixels along width, n: no. of pixels along height }
Pixel_xy ← collinearity(params, imageDimension)
for i in |pixel_xy| **do**
    x_i ← m * 0.5 + pixel_$xy_{(i,x)}$
    y_i ← n * 0.5 + pixel_$xy_{(i,y)}$
end for
isInPicture ← inPicture(x, y, image_dimensions)
for i in |imageDimensions| **do**
    $index_i$ ← i * (isInPicture > 0)
end for
for i in |index| **do**
    n_i ←-- distance (x_s, y_s, z_s, $t_{(i,x)}$, $t_{(i,y)}$, $t_{(i,z)}$)
    x ← x_{index}
    y ← y_{index}
    t ← t_{index}
end for

for i in |index| **do**
    if n_i > 50 AND n_i < [distgrid, xgrid, ygrid]
        distgrid, ← n
        xgrid ← $t_{(i,x)}$
        ygrid ← $t_{(i,y)}$
    end if
end for
return [distgrid, xgrid, ygrid]

Photogrammetry. The model used to project the 3-D model data onto a two dimensional image used photogrammetry techniques. The Algorithm 1 summarizes this process. The model takes as inputs seven camera parameters:

1. X_s : x coordinate of camera (longitude)
2. Y_s : y coordinate of camera (latitude)
3. Z_s : z coordinate of camera (elevation)
4. K : roll of the camera
5. Φ: pitch of the camera
6. Ω : yaw of the camera
7. F : focus of the camera

Algorithm 2. Collinearity algorithm.

Inputs: t= {$t_{..x,y,z..}$, $t_{..x,y,z..}$, $t_{..x,y,z..}$, … } :a numpy array representing x,y and z for each pixel of the tile
Parameters: {Params = κ: roll, φ : phi, ω : omega, x_s : longitude, y_s : latitude, z_s : elevation, f : focus}
c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9 = scalingFunction(κ, φ, ω. x_s, y_s, z_s, f)
For i = 1 to | t | **do**
    $y_{num(i)}$ ← c_4 * ($t_{(i,x)}$ -x_s) + c_5 * ($t_{(i,y)}$ - y_s) + c_6 * ($t_{(i,z)}$ - z_s)
    $x_{num(i)}$ ← c_1 * ($t_{(i,x)}$ -x_s) + c_2 * ($t_{(i,y)}$ - y_s) + c_3 * ($t_{(i,z)}$ - z_s)
    $denom_{(i)}$ ←- c_7 * ($t_{(i,x)}$-x_s) + c_8 * ($t_{(i,y)}$- y_s) + c_9 * ($t_{(i,z)}$ - z_s)
    xx ←- - f * $x_{num(i)}$/ $denom_{(i)}$,
    yy <- f * $y_{num(i)}$/$denom_{(i)}$
end for
Return : x, y pixel location for the 3d model

Algorithm 3. Checking if pixels are in extent of image.

Input: x : {x_1, x_2, x_3, … } : pixel along width of image, y : {y_1, y_2, y_3,….} : pixel along height of image
Parameters: m : no. of pixels along width, n: no. of pixels along height
For i in |x| **do**
    pixelnImage$_i$ ← (x_i < m) AND (x_i > 0) AND (y_i > 0) AND (y_i < n)
End for
Return: pixelnImage
Output : Boolean array indicating pixels which are part of the image

Once these parameters are obtained, these parameters along with the image dimensions are passed into a function which converts the 3-D data of each tile into two dimensional pixel information. This transformation is done by using a transformation matrix containing the seven degrees of freedom

of the camera as seen in Algorithm 2 in the collinearity function.

The output from Algorithm 2 containing the possible pixel coordinates is then mapped into the actual image plane (plane of the given image) using the *isInImage* function. This function is explained in Algorithm 3. This is done to identify if the mapped information lies in the plane of the given image. The algorithm checks if the pixels of the projection lie within the extent of the actual image. This function outputs those points which are successfully projected from the tile on the image plane. Once the actual projections from the tile are obtained the longitude, latitude and the distance of the location projected on each pixel is stored in a list of three arrays: *distgrid*, *xgrid* and *ygrid*. The distance of the location of the pixel and that of the camera is obtained using Euclidian distance methods.

Algorithm 4. Identifying correct values for the pixels.

```
Input: currentVal(distgrid, xgrid, ygrid), newVal(distgrid, xgrid, ygrid)
if currentVal(distgrid) < newVal(distgrid)
        replace ← True
else
        replace ← False
end if
output(distgrid, xgrid, ygrid) ← currentVal(distgrid, xgrid, ygrid) * (NOT replace) + newVal(distgrid, xgrid, ygrid) * replace
return output
Output: list containing distgrid, xgrid, ygrid
```

The process in Algorithm 1 is repeated on each tile. Once this process is completed in order to identify the correct project Algorithm 4 is implemented. This algorithm compares for each pixel the *distgrid* values and stored information for only the smaller of the two values. This is done repeatedly for each tile till we get the appropriate information. Once the process is complete after comparing the pixels for every tile, the final output is returned as three files which contain the euclidian distance of the points for each pixel, the longitude information for each pixel and the latitude information for each pixel.

C. Implementation in Spark

We implemented Algorithms 1 and 4 in Spark. We used the NYU Dumbo which is a 48-node Hadoop cluster, running Cloudera CDH 5.9.0 (Hadoop 2.6.0 with Yarn).

The spark version we used is Spark 1.6 with module "python/gnu/2.7.11". The pyspark script and the numpy files were uploaded to Dumbo using WinSCP.

The numpy files were read into RDD as binary files and a mapPartitions function was used to implement Algorithm 1. The treeReduce function was used to implement the Algorithm 4. The script was submitted using following command:

```
spark-submit --num-executors 5
--conf spark.port.maxRetries=50
--conf spark.kryoserializer.buffer.max=1024
--conf spark.dynamicAllocation.enabled=false
--conf spark.driver.memory=4G
--conf spark.driver.cores=4
--conf spark.executor.memory=4G hadoop_script.py
```

The average runtime for the script with the above configurations was about 1min 34 seconds

D. Post processing

While performing the photogrammetry there is a possibility of points on the image not being touched by any point on the 3D model. If these points lie on building surfaces displayed in the image there might be continuity errors on the image. In order to minimize these continuity errors we perform a smoothing function called cascading over the output data. While performing the cascading an important assumption of "no overhangs" has be taken into consideration. This assumption ensures that if a pixel has default value then every pixel having a higher y coordinate in the image plane (height) has the same default value. If not then every point below that higher y coordinate (which has a non default value) is converted replaced with the value of the higher y coordinate point. The post processing image is shown in Figure 4.

III. Challenges

A. Variety

One of the key challenges we encountered during this project had to do with different data types and data formats. First, the 3-D building model data was contained in two different formats: geolocations were stored in Multipatch (ESRI) and geometries in DGN (Microstation). As were also working with image data, we faced a challenge of integrating unstructured and structured data.

B. Volume

After converting raster files into .npy format as described earlier, the size of the 3-D dataset increased to almost 9 GB. Processing these data required parallelization and high-performance computing resources.

C. Processing complexity

Memory. Spark runs the script at the driver as well as the executors. The script has a lot of variables which are created at runtime. So we had to increase the heap memory at the driver as well as the executor so that it could handle the data size. We set our driver and executor memory size to 4 GB each.

Number of cores. The number of concurrent tasks that run at the executor level is equal to the number of cores. We set the number of cores to 4 for the driver.

Kryo serialization. Spark in Dumbo is using Kryo Serializer to convert between the deserialized Java object representation and the serialized binary representation. [6] The size of objects created in our code is large so we had to increase the max size

of the kryoserializer buffer to 1024 MB.

Dynamic allocation and number of executors. Spark 1.6 has an option to assign and remove executors as per requirement. However, there is no definite way to decide whether an executor will run a tasks in the future. There are different ways to implement this, e.g. using external shuffler. We decided not to use dynamic allocation of resources and configured it to false. As per Cloudera "how to tune your apache spark jobs" blog we can get a full write throughput by using five tasks per executor. So we set the number of executors to five.

D. *Scalability*

Extending field of view from Lower Manhattan to the rest of the city as well as achieving a higher resolution than one foot would require additional computing resources.

IV. Outcome

The output from the MapReduce phase is visualized in Figure 3. As a starting value, all distances for all pixels were set to 100,000 ft. Every time one of the points in the 3-D model (tiles) was projected to one of the pixels, the output triple of arrays (each one with the same dimension of the picture) stored the distance from that point to the camera, the latitude from that point, and the longitude.

After the result obtained from the MapReduce phase (Figure 3), the cascading algorithm was applied, assuming that there were no overhang structures in the field of view, to get a full solid representation of the buildings in the image (Figure 4).

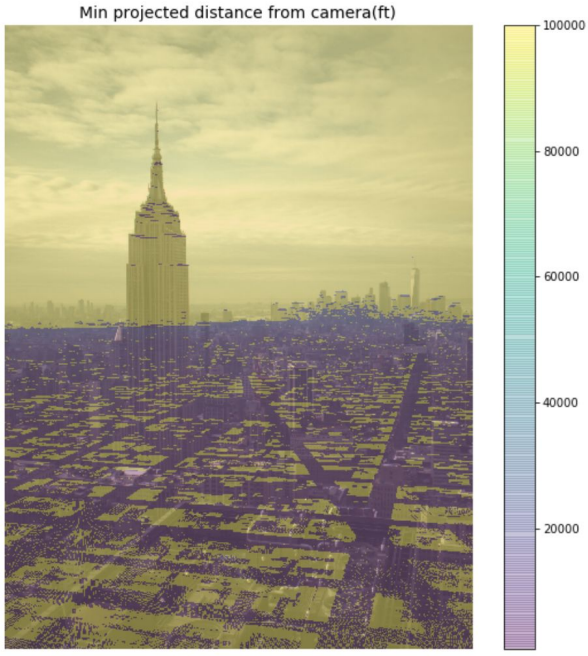

Figure 3. Representation of the minimum distance projected to each of the pixels in our image. Yellow pixels are the ones that had no points projected onto them, while purple pixels are the ones that had projections from the 3-D model.

Figure 4. Result of the cascading algorithm. The color coding goes from red to blue in an infinite cycle. In the lower part of the images, there is not a perfect match because when projecting the points from the 3-D model to the picture, it was included a constraint of only projecting the points that were at least 500 ft away from the camera. Because of that constraint the big building in the bottom of the image was not projected. Therefore, the colored representations of buildings visible in the image are actually covered in the real image by that closest rooftop. It is also noteworthy that since the Brooklyn tiles from the 3-D model were not included in the analysis, the building from that part of the city were not identified in this method.

Afterwards, since the latitude and longitude information of the point projected in each pixel was also stored in the output, it was possible to get from MapPLUTO the BBL numbers for each of those buildings.

References

[1] T.Schenk. Introduction to Photogrammetry. Ohio State University: 2005.

[2] NYC Department of Information Technology and Telecommunications. NYC 3-D Building Model. http://www1.nyc.gov/site/doitt/initiatives/3d-building.page

[3] The project referred extensively to and adapted existing UO code: https://github.com/gdobler/nycene

[4] All project code is saved in the following Github repo: https://github.com/vishelar/Big_Data_UO_Ray_Tracing

[5] Cloudera. How-to: Tune Your Apache Spark Jobs. http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/

[6] https://spark.apache.org/docs/latest/job-scheduling.html