CSUMB Capstone Fall 2020
Sean Towne & Edward Montoya
STOQS Github issue #1093

Introduction

This report is a summary of the work we did during our fall 2020 capstone project. It is also meant to be a potential resource for anyone interested in working on STOQS Github issue #1093. Our original goal for our capstone project was to update the terrain generation pipeline used by STOQS to make use of a newer and faster file type, glTF. The scope of our project quickly changed to simply converting the bathymetry data of the Monterey Bay into a Wavefront OBJ, an intermediate step towards converting it to a glFT file, due to the complexity of the task. The rest of this report attempts to bring newcomers to the problem up to speed with it, walk them through what we tried, and demonstrate what finally worked for us.

GMT and MB-System

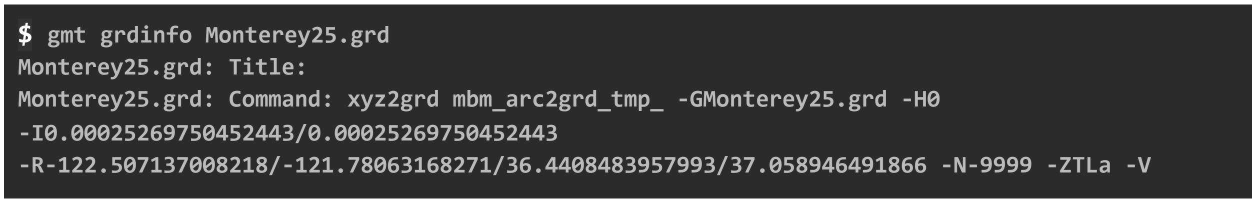
Our capstone mentor Mike McCann suggested that we use tools from GMT and MB-System to work with the geographic data of the Monterey Bay. GMT stands for generic mapping tools, and is a software suit used by researchers to work with geographic and bathymetric data. MB-System is a portion of GMT contributed by engineers at MBARI. We make many references to tools in this suit in this report and their documentation can be found here.

Monterey25.grd

Monterey25.grd is a binary file that contains, along with a small amount of metadata, millions of points along the ocean floor of the Monterey Bay. The points are in units of latitudinal and longitudinal degrees, and meters in relation to sea level for elevation. A summary of the Monterey25.grd file can be obtained by using the `grdinfo` command from GMT.

```
$ gmt grdinfo Monterey25.grd
```

This command will print out useful information about the file, including the command used to generate it in the first place. We used it to obtain the number of rows and columns of points in the file, as well as some other values that we used as arguments in other GMT commands. The points in the file represent a rectangular geographic area, and are arranged in the file such that the first C points represent the first row, the second C points represent the second row and so on, where C is the number of columns of data. The first point in the file is the “top left” point in the rectangle and the last point in the file is the “bottom right”.



```
$ gmt grdinfo Monterey25.grd
Monterey25.grd: Title:
Monterey25.grd: Command: xyz2grd mbm_arc2grd_tmp_ -GMonterey25.grd -H0
-I0.00025269750452443/0.00025269750452443
-R-122.507137008218/-121.78063168271/36.4408483957993/37.058946491866 -N-9999 -ZTLa -V
```

```
Monterey25.grd: Remark:
Monterey25.grd: Gridline node registration used [Cartesian grid]
Monterey25.grd: Grid file format: cf = GMT netCDF format (32-bit float, deprecated)
Monterey25.grd: x_min: -122.507137008 x_max: -121.780631683 x_inc: 0.000252697504525 name:
user_x_unit n_columns: 2876
Monterey25.grd: y_min: 36.4408483958 y_max: 37.0589464919 y_inc: 0.000252697504524 name:
user_y_unit n_rows: 2447
Monterey25.grd: z_min: -3012.47290039 z_max: 964.723083496 name: user_z_unit
Monterey25.grd: scale_factor: 1 add_offset: 0
Monterey25.grd: format: classic
$
```

A problem with the data in the Monterey25.grd file is that it is in the wrong units for computer graphics rendering. Computers work much better with points in a three dimensional x,y,z space. To do a transformation from geographic coordinates to a three dimensional space we can use the mapproject program from GMT.

GMT Mapproject

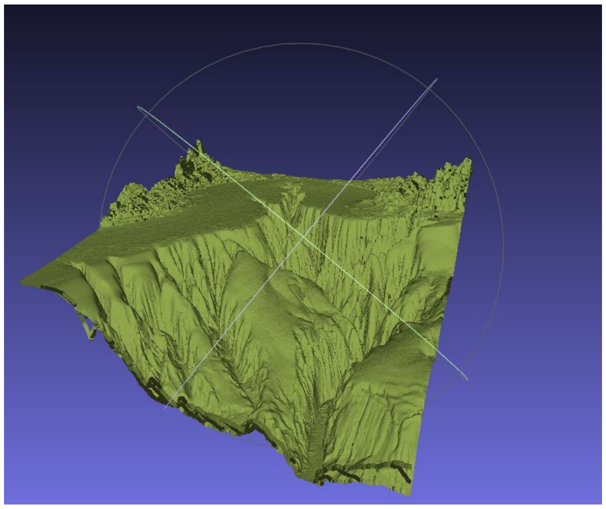
Mapproject takes as input geographic coordinates like those in Monterey25.grd in an ASCII table format and projects them to various coordinate systems. Because we are working with geographic coordinates and want them to be in a Cartesian space, we can use the -E flag from mapproject to transform the points in Monterey25.grd to x, y, and z triplets with the origin at the center of the earth, and the z axis going through the poles. Here is an example usage, omitting some preprocessing of Monterey25.grd.

```
$ cat preprocessedMonterey25.txt | gmt mapproject -E > pointCloud.asc
```

This usage would produce a file called pointCloud.asc that contains lines of x, y, z triplets, where each line corresponds to a latitude, longitude and altitude triplet in Monterey25.grd. In Meshlab, this ASCII point cloud can be imported and it renders nicely, especially after using the Meshlab function: *filters -> point sets -> compute normals for point sets.*

Before computing point normals

After computing point normals



The crux of our project from this point was to convert this point cloud to an OBJ with all the necessary meshlab processing of it represented in the OBJ file, and to do so in an automated fashion. Meshlab is capable of exporting this point cloud as an OBJ file, but it is still necessary to import the point cloud, perform several operations on it and export it manually. A solution to this problem is discussed later in this report.

Wavefront OBJ files

OBJ files are used in computer graphics for rendering complex shapes and are useful because they make it easy to specify relationships between points, i.e. polygons. An excellent and brief resource for understanding OBJ files can be found here. In short, an OBJ file is a list of points, called vertices, normal vectors, and polygons, called faces. A point's normal vector can be inferred from the faces it is part of, so it is not always necessary to include. Below is an example of a very simple OBJ file that defines a triangle.

```
# triangle.obj
# vertices
v 0 0 1
# vertex normals can be listed as: vn x y z
v 0 1 0
v 1 0 0
# faces
f 1 2 3
```

OBJ files are well supported and well known, and it is very likely that any pipeline to convert geographic data to a glTF format would convert to OBJ at some point.

Converting to OBJ

Below we enumerate some of the strategies we investigated to convert our point cloud of the Monterey Bay to a Wavefront OBJ file. The final section *Automating Meshlab and exporting an OBJ* elaborates on the approach we had the most success with.

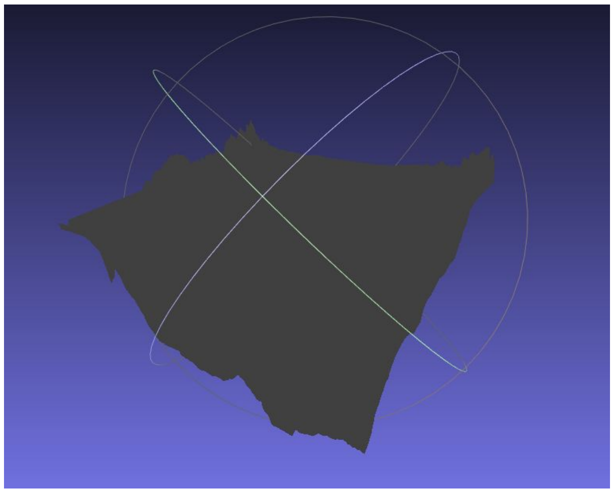
A naive approach

A naive approach to converting a point cloud to an OBJ format would be to write a script that simply loads the point cloud data into memory as a grid of points, and iterates through the grid creating a list of simple polygons. Below is an attempt of ours to do so. It loads in a point cloud, writes each point to an OBJ file with a ‘v’ prefix, and iterates through the grid writing sets of points prefaced with an ‘f’. The sets of points define a square rather than a triangle, that is for simplicity's sake.

```
#! /usr/bin/python3

import matplotlib.pyplot as plt
import numpy as np
file = open("test.asc", 'r')
raw_data = file.readlines()
output = ""
COLUMNS = 2876
ROWS = 2447
grid = np.full((ROWS, COLUMNS, 4), 0.0)
k = 0
for i in range(ROWS):
	for j in range(COLUMNS):
		nextLine = raw_data[k].split()
		x, y, z = nextLine
		output += "v "+nextLine[0]+" "+nextLine[1]+" "+nextLine[2]+"\n"
		grid[i][j] = [float(x), float(y), float(z), float(k+1)]
		k += 1
output += "\n"
for i in range(ROWS-1):
	for j in range(COLUMNS-1):
		v1 = str(int(grid[i][j][3]))
		v2 = str(int(grid[i+1][j][3]))
		v3 = str(int(grid[i+1][j+1][3]))
		v4 = str(int(grid[i][j+1][3]))
		output += "f "+v1+" "+v2+" "+v3+" "+v4+"\n"
output_file = open("test.obj", "w")
output_file.write(output)
output_file.close()
```

This produces a valid OBJ file that loads into Meshlab however, the normals of each square were not inferred by meshlab, counter to our expectation. The resulting OBJ file looks like this.



It is very much like our point cloud before computing normals. A glaring problem with this file is that there are too many faces, Meshlab takes a very long time to import it, and to rotate it. The file is too big. Another problem with this naive approach is that in order to simplify the code, we included NaN values from the original Monterey25.grd file, you can't see the effect in this image, but they produce lines along the edges of the image near the sea level altitude. We believe this is an artifact of how the data was collected.

The little python program above is a proof of concept. We did not have high hopes for manually performing this step in custom code, but this program proves that at least the basic operations of converting a point cloud to an OBJ is trivial. Certainly, anyone who would want to pursue this approach further would need to consider sampling points from the point cloud, as the number of points in the file are simply too numerous. It would also be necessary to compute the normal vectors for each point sampled, and to compute what triangle each point should be a part of. We expect that the grid-like nature of the point cloud should make the triangle creation easier. Many surface reconstruction algorithms can't assume gridded data and are therefore much more complicated. As to computing normal vectors, we do not know how to do that. One approach could be to assign the normal vector of a triangle to each of its points, but most points would be associated with more than one triangle, we do not know how to resolve that ambiguity. However, solutions to these problems are certainly out there on the internet, and we think that taking this route to convert a point cloud to an OBJ file is certainly the most exciting way to do it. Unfortunately, the complexity of this route put it out of our grasp due to our time constraints.

Using GMT and MB-System Tools

In MB-System there is a fairly new program called mbgrd2obj that reads in .grd files and outputs graphics friendly OBJ files. We were not able to get it to work with our Monterey25.grd file. This is most likely due to an issue with a suspected mismatch between the units mbgrd2obj expects and the units used in Monterey25.grd. We have some evidence that this is the case because we did some work to rule out another possible problem: that NaN values in Monterey25.grd were causing mbgrd2obj to output garbled data.

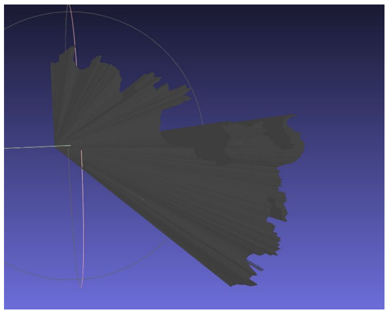
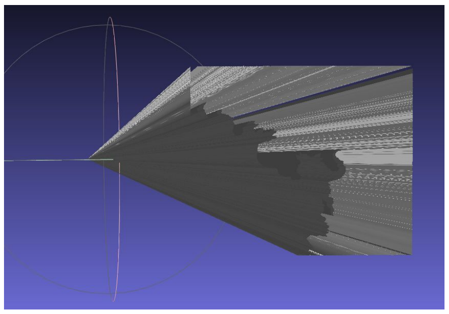
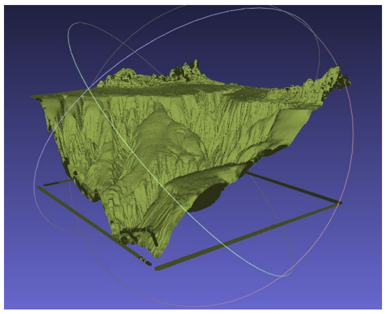
To do this we substituted all of the NaN values in Monterey25.grd with the smallest elevation value found in the data set. We chose the smallest value because we needed a valid elevation value, and because points with the smallest altitude are conspicuous, as we will see. All of the NaN values in Monterey25.grd are missing altitude values. That can be demonstrated with:

```
        gmt grd2xyz Monterey25.grd --FORMAT_FLOAT_OUT=%f | awk '{print $3} | grep NaN | wc -l
        gmt grd2xyz Monterey25.grd --FORMAT_FLOAT_OUT=%f | awk '{print $1, $2}' | grep NaN | wc
-l
```

Which return 126214 and 0 respectively. So if we impute them with a `sed` replacement we can rest easy knowing that our longitude and latitude values are unchanged. We can replace all the NaN values in Monterey25.grd with a valid elevation (smallest elevation) and re-encode as a grd with:

```
        gmt grd2xyz Monterey25.grd --FORMAT_FLOAT_OUT=%f | sed
        's/NaN/-3012.47290039/' | awk '{print $1, $2, 10 * $3}' | gmt xyz2grd
        -GNoNaN.grd -I0.00025269750452443/0.00025269750452443
        -R-122.507137008218/-121.78063168271/36.4408483957993/37.058946491866
```

This command produces a file called NoNaN.grd that would be identical to Monterey25.grd except that all NaN altitudes are replaced with the smallest altitude. The options passed to xyz2grd come from the metadata in Monterey25.grd. When NoNaN.grd is fed to mbgrd2obj, the result is the same as feeding Monterey25.grd to mbgrd2obj, except with the conspicuous lines created from replacing NaN values with small but valid altitude values. Below are depicted the outputs of mbgrd2obj when ran against Monterey25.grd and NoNaN.grd respectively. The third graphic represents closer to what we had hoped mbgrd2obj would produce when ran against NoNaN.grd, it is just a point cloud generated from running mapproject against the NoNaN data.

These pictures indicate to us that the problem with mbgrd2obj is indeed with the units and not caused by mishandling of NaN values in Monterey25.grd. One important thing to note is that the data in NoNaN.grd that is piped through mbgrd2obj has not undergone projection by mapproject. We had hoped that mbgrd2obj would do something similar to mapproject since it seems the goal of mbgrd2obj is to make geographic data graphics friendly.

We considered piping our data through mapproject to ensure that it is in an earth centered earth fixed x, y, z coordinate system, converting it back to a grd file with xyz2grd and then trying the projected data with mbgrd2obj. The problem with this is that mapproject destroys the gridded nature of our data, the x, y, and z columns no longer correspond to uniform latitude and longitude values. Think of a plane in a 3d space that has been transformed away from the origin, and rotated about multiple axes. After using mapproject all the relative positions that the points represent are maintained, but the numbers used to describe the points are emphatically, not geographic coordinates. This is why a pipeline that looks like this would not work:

```
Grd2xyz -> mapproject -> xyz2grd -> mbgrd2obj
```

We considered that maybe a properly transformed and rotated point cloud from mapproject might work with mbgrd2obj. The trouble would be converting such a point cloud back into a grid file. If the point cloud represented z values as a function of gridded x and y values, then it might work with xyz2grd. The point cloud produced by mapproject does not represent z values as a function of gridded x and y values, rather, no axis represents elevation, and the grid lines do not align with any axis.

There certainly exist matrices that could rotate the point cloud such that the z value in each point could be thought of as a function of the other two points. A transformation (in the mathematical sense) could also be performed to move all the points closer to the origin. This could be useful for some workflows, and the transformation would set the point cloud up to work better with graphics processors down the line as well.

Automating Meshlab and exporting an OBJ

We finally had some success after finding a tool called MeshlabServer. It is a command line tool that comes packaged with Meshlab. MeshlabServer uses an XML like file to read in meshlab commands that it then runs against an input file specified as an argument, and writes the resulting mesh to an output file, also specified as an argument. Meshlab supports Wavefront Obj files, and the MeshlabServer has no problem saving its output as such. The Meshlab GUI itself keeps a list of filters applied to a mesh and automatically puts together a markup file that can be used with MeshlabServer. Below is a small tutorial/demonstration of what we did.

In Meshlab you can load in the point cloud that is produced by mapproject and apply various filters like computing normals for point sets, laplacian smoothing, etc. Once the mesh is looking the way you want it, you can select:

```
Filters -> Show current filter script -> save
```

Meshlab will save the file as a .mlx. This file contains all the filters you applied to the point cloud and their parameters, and the exact workflow you just did in the GUI can be reproduced with this script and MeshlabServer. Some tags in the file contain attributes that serve more as comments and just make the file hard to read, they can be removed with:

```
sed -i 's/tooltip="[^"]*"//' poissonOriginal.mlx
sed -i 's/description="[^"]*"//' poissonOriginal.mlx
```

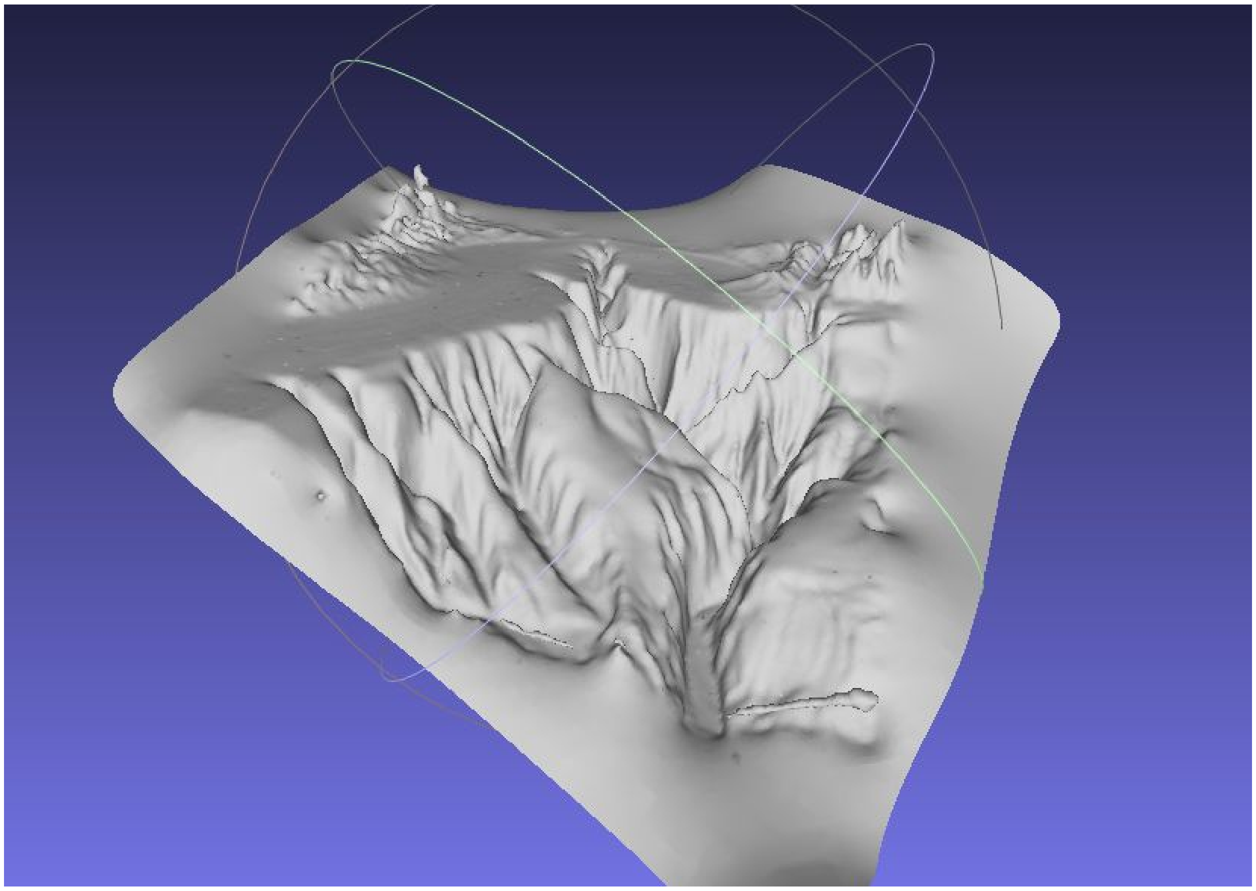
After saving the script from meshlab and running these commands we get a file that looks like this:

```
$ head filterScript.mlx
<!DOCTYPE FilterScript>
<FilterScript>
 <filter name="Compute normals for point sets">
  <Param type="RichInt" value="10"   name="K" isxmlparam="0"/>
  <Param type="RichInt" value="0"   name="smoothIter" isxmlparam="0"/>
  <Param type="RichBool" value="false"   name="flipFlag" isxmlparam="0"/>
  <Param type="RichPoint3f"   z="0" x="0" name="viewPos" isxmlparam="0" y="0"/>
 </filter>
 <xmlfilter name="Surface Reconstruction: Screened Poisson">
  <xmlparam value="0" name="cgDepth"/>
. . .
. . .
```

From this point, generating an OBJ with all the filters we want applied to it is as easy as invoking MeshlabServer. If our point cloud data is saved as pointcloud.asc and our filter script is saved as filterscript.mlx, and we want our output to be mesh.obj, then we could use a command like this:

```
meshlabserver -i pointcloud.asc -o mesh.obj -s filterscript.mlx
```

We ran this command with a script that followed as closely as we could get to the steps defined in the pipeline on github. We used a “screened poisson” filter where the pipeline uses a “poisson” filter. The arguments to each filter are different, so we went with the default arguments. We did this because we could not find the exact filter listed in the pipeline. Our result looks like this:



As you can see, there is some residual surface left by the surface reconstruction algorithm “screened poisson”, and some loss in precision. We are not offering this as a final solution to this part of the STOQS terrain generation pipeline, rather as a proof of concept. The meshlab procedures outlined in the STOQS documentation were performed with an older version of Meshlab, whereas the only versions we could find on the internet (this was really strange, we could not find version 1.3.3 anywhere) are versions from the last couple years. We obtained this version number from running `MeshlabServer -h`

```
MeshLabServer version: 2020.03+dfsg1
```

So the remaining work to be done to finish automating the meshlab portion of the terrain pipeline involves experimenting with the filters in newer versions of Meshlab in order to replicate these steps from the old version:

```
File -> Import Mesh...
 Filters -> Point Sets -> Compute normals for point sets (defaults)
 Filters -> Remeshing, Simplification and Reconstruction ->
           Surface Reconstruction: Poisson (Octree Depth: 12, Solver Divide: 10)
 Filters -> Remeshing, Simplification and Reconstruction -> Quadric Edge Collapse
Decimation:
     - Target number of faces: 1,500,000
     - Preserve Normal
```

```
    - Preserve Topology
    - Optimal Position of Simplified Vertices
    - Planar Simplification
    - Post-simplification cleaning
Filters -> Normals, Curvatures and Orientation -> Invert Faces Orientation (if needed)
* Cleanup - use editing tools (with plenty of intermediate saves)
Filters -> Smoothing, Fairing and Deformation -> Laplacian smooth (surface preserve)
```

Conclusion

We set out at the beginning of the semester to update the terrain generation pipeline and build support for glTF files in STOQS, but had to quickly scale back our ambitions when we realized how complicated the problem of working with terrain data can be. We explored the tools from GMT and MB-System, and unfortunately were not able to make much use of them. It should be stated that those tools are highly specialized and someone with more experience in computer graphics, mathematics, and cartography should still consider investigating them. Just because we couldn't get them to work does not mean they cannot solve this problem.

We wrote a small python program that took points from our cartesian point cloud and listed them out as vertices and faces in OBJ format, and the resulting OBJ file was valid and loaded into meshlab. However, there is more to making a good looking and usable mesh than that. Unfortunately, constrictions on our time meant that we could not pursue that route any further, despite the fact that it is one of the more alluring options because of its engineering appeal.

Finally, we discovered that meshlab is bundled with a command line interface called MeshlabServer that we used to successfully automate the meshlab portion of the STOQS terrain generation pipeline and export the mesh as a Wavefront OBJ file. We did not have time to experiment with the filters in our newer version of Meshlab, so, in order to have an OBJ file that looks as good as and performs like the .ply file in the old pipeline, someone will still have to go into meshlab and find what procedures will produce and acceptable mesh.