Shairoz Sohail Follow

Feb 28, 2022 · 9 min read · Listen

Save

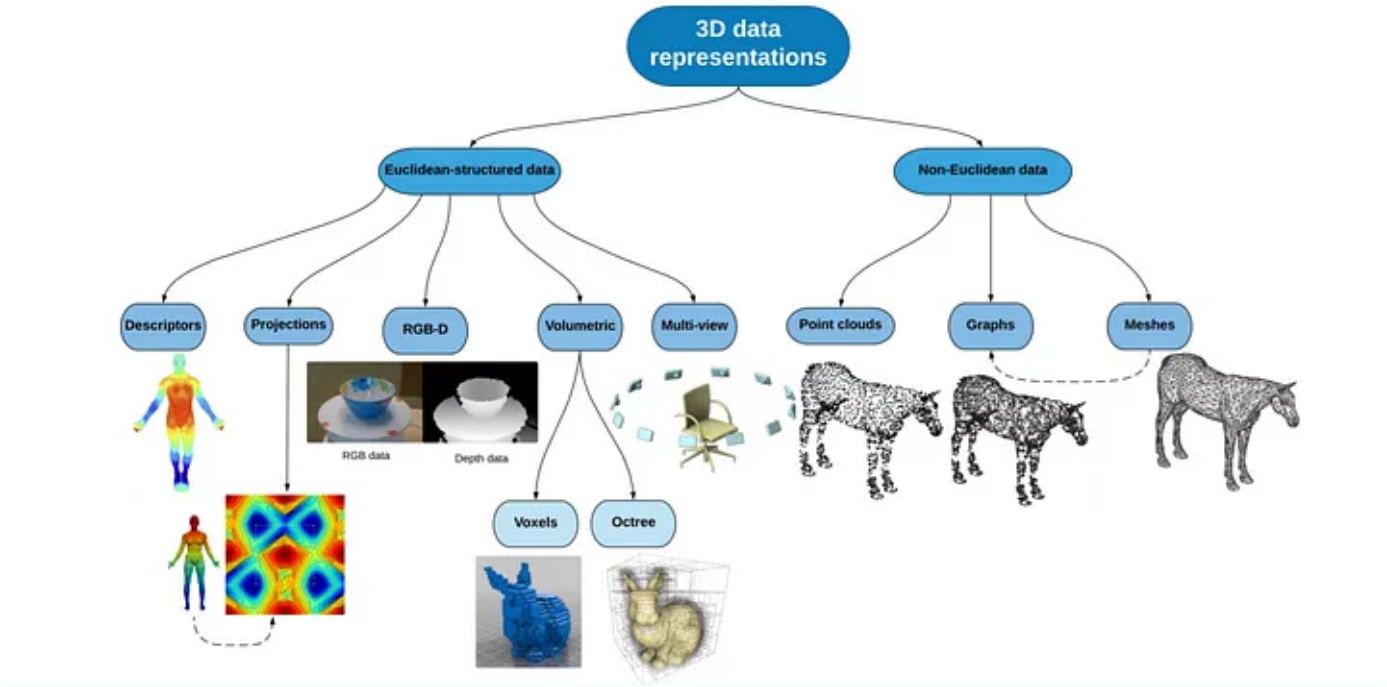
Generating 3D Models with Deep Learning (part 1)

The digital world is fast growing an extra dimension, with the advent of technologies like virtual reality and autonomous vehicles, the systems of the future will begin to see the world just the way we see it: with three dimensions. Someday staring at a 2D picture on a screen may become no different than using a paper map.

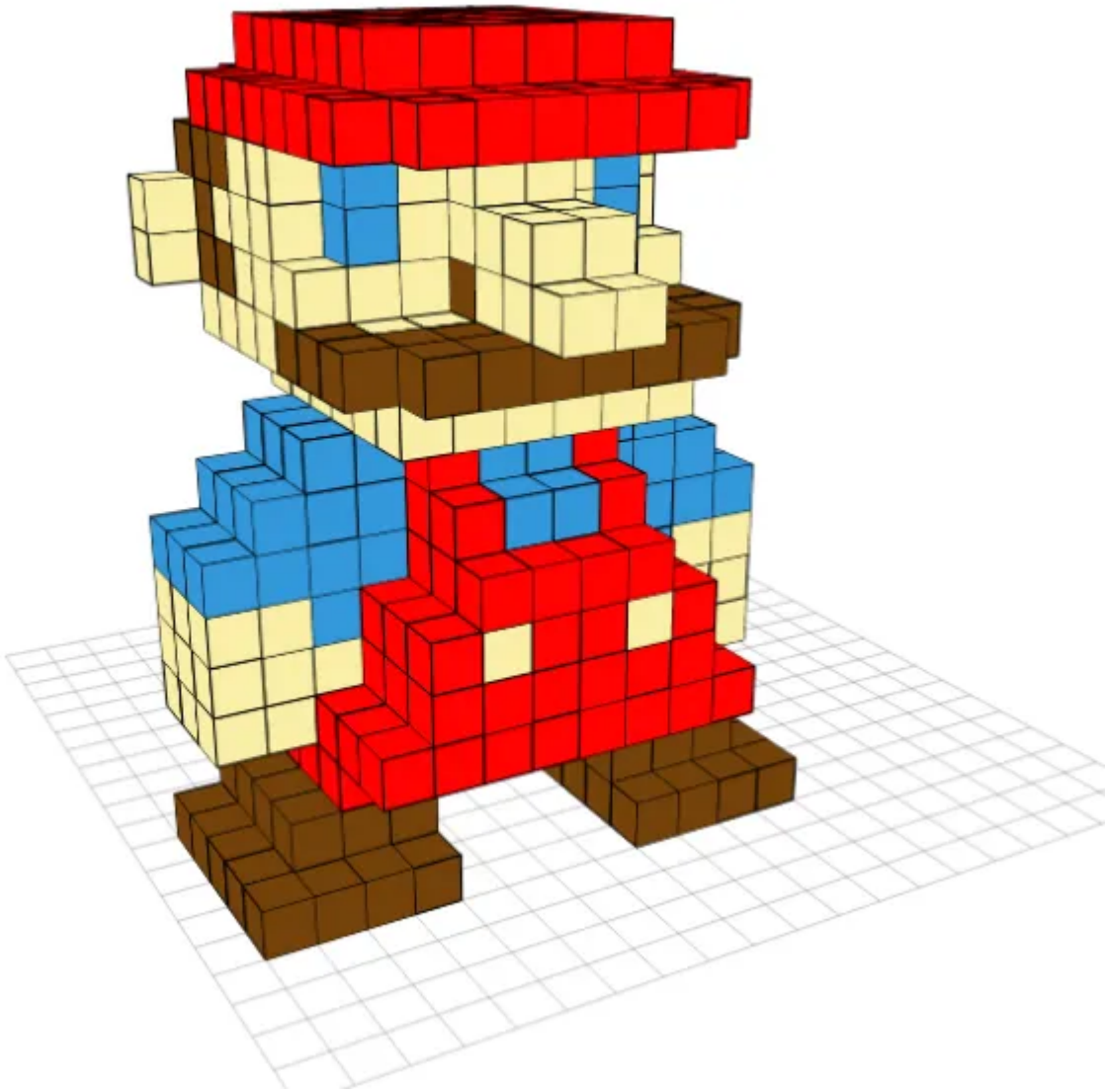
A huge part of this shift into 3D is object models. Representations of real or imagined objects in their full geometric glory, able to be rotated and viewed in all sorts of angles, imposed with different surface textures, and able to be interacted with in all kinds of interesting ways. Think furniture in the metaverse, or a CAD model of the hose on your vacuum cleaner, whatever floats your boat.

There is a huge need for these models, while complex 3D scenes include all kinds of auxiliary information about terrain and lighting and texture, the scenes are barren without actual objects. The sources for these 3D objects are hugely varied, from LiDAR to 3D scanners to photogrammetric reconstructions, all of these methods are relatively time consuming and fairly expensive — the opposite of where 2D imaging technology

sits right now. Before we get into the ways we can collect "true" 3D data, we must differentiate between euclidean and non-euclidean formats, and differentiate between 3D and 2D+ formats. The beautiful illustration below (taken from this paper), shows the various formats:



euclidean-structured data has a common system of coordinates, i.e we can refer to the depth at (x,y) for two separate RGB-D images. We can do the same with volumetric data such as voxels by referring to the corresponding grid number.



Mario reduced to a mere Voxel representation of himself

From a standpoint of data volume, there is significantly more 3D euclidean-structured data available vs. non-euclidean (RGB+D datasets especially). This is mainly because of the cost and time associated with creating authoritative non-euclidean datasets. For example, an RGB+D dataset can be captured with a Kinect device, which is under $300. Compared to a similar use LiDAR scanner from Einscan that is priced at $6,899 and it becomes clear why it is difficult to get a hold of non-euclidean datasets.

Mario in full 3D glory

For the remainder of this article, we will talk primarily about non-euclidean datasets, which come primarily from one of two sources:

LiDAR

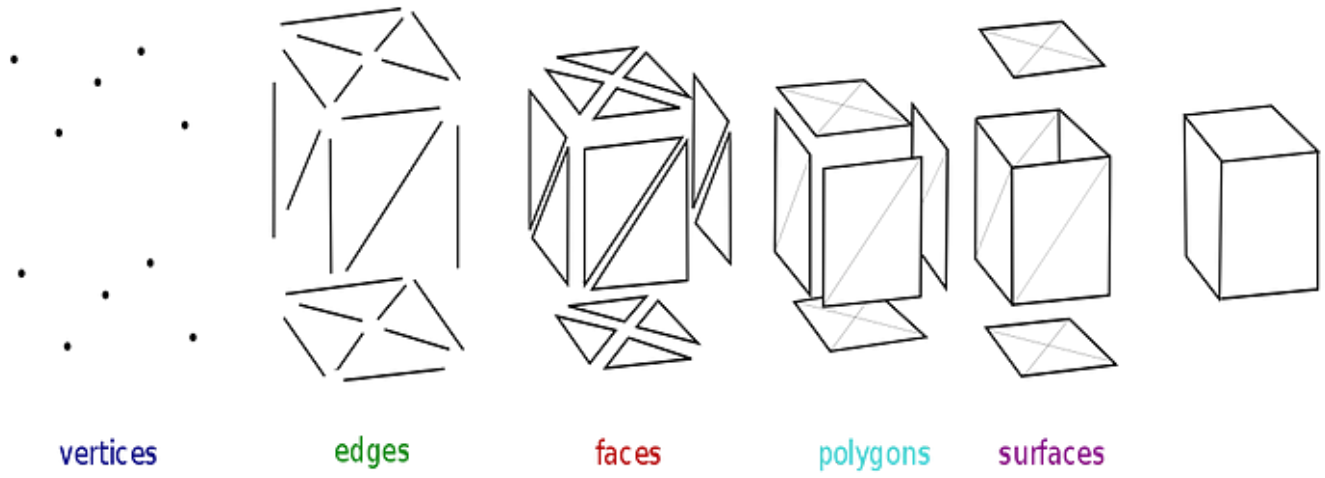
Explaining the full particulars of how LiDAR sensors works is beyond the scope of this article, but the general idea is shooting a laser at an object and measuring the return — doing this repeatedly allows you to map the general surface topology of an object or scene. Commercial LiDAR is typically mounted either on vehicles or an aircraft, known as *terrestrial* and *aerial* LiDAR, respectively. Outside of large scale commercial and government use, LiDAR is generally out of price range for most people.

It is important to state however, that with recent technological advancements it seems that lower power/resolution LiDAR scanners are finally making their way into consumer devices, such as the scanners found on recent flagship iOS devices like the iPhone and Ipad Pro.

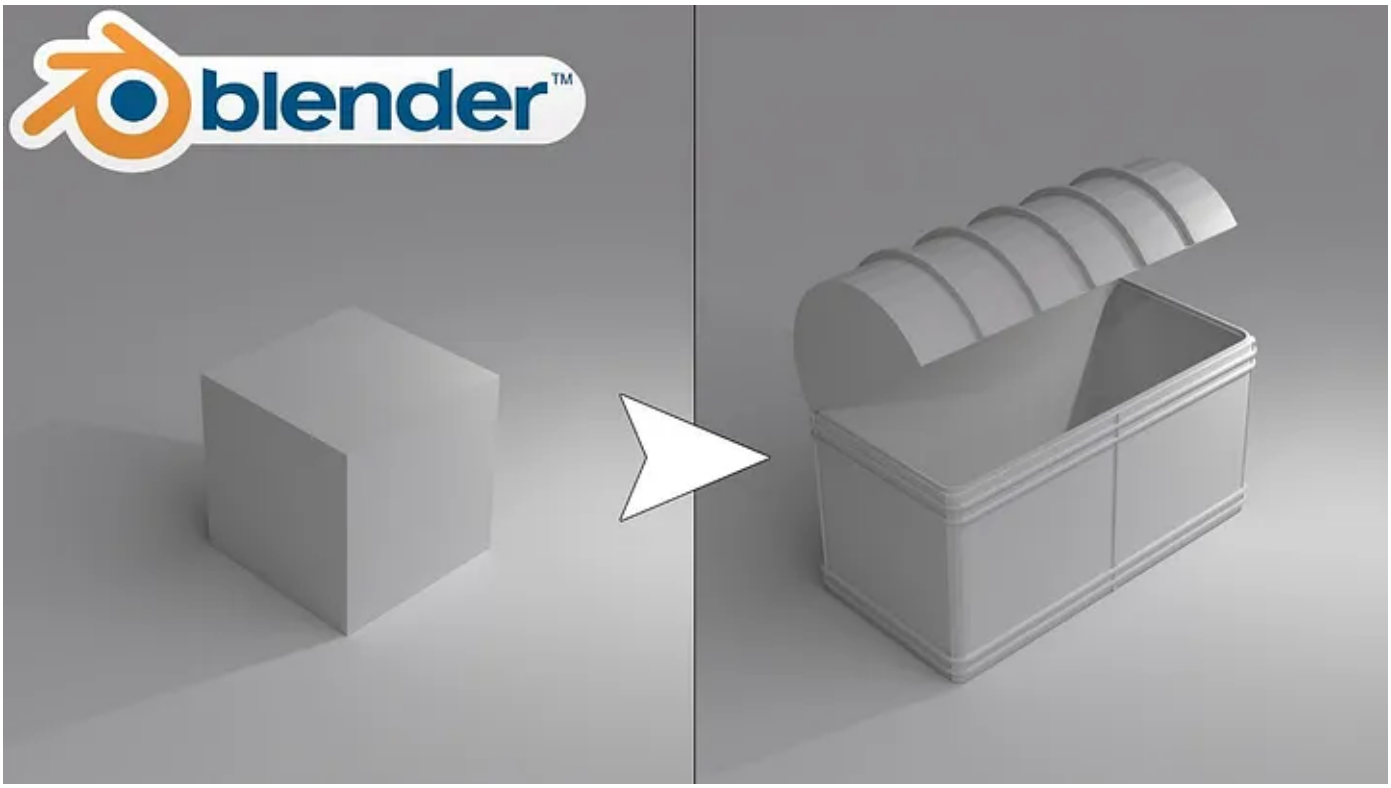
Scanned with a Iphone 12 Pro — good for single objects, not great for large complex scenes

Mesh models

Object meshes generally refer to polygonal meshes tessellating the surface of an object. These are composed of the following elements:



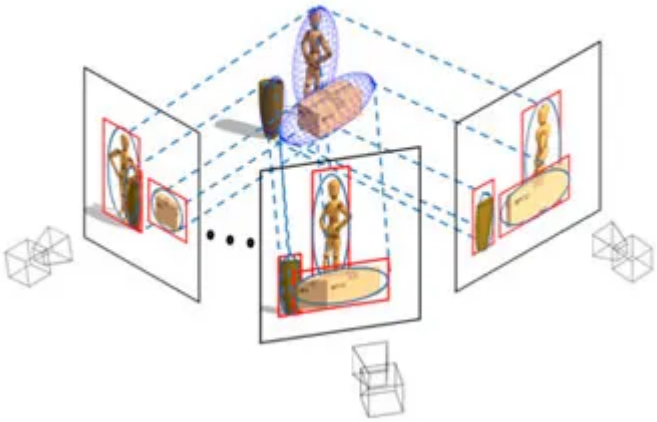
These are typically created by computer graphics modeling using one of the many 3D modeling packages in circulation. Generally, the artist uses an actual object as reference, and either gradually adds vertices and faces to morph a primitive object such as a cube/pyramid/cylinder into the desired shape, or by drawing a 2D top-down view and extruding using additional images for depth reference.



Tutorials — blender.org

Bonus: Meshes from Photogrammetry

Humans see in 3D but are not equipped with any kind of radar-based sensing system (unlike bats). This is because having a pair of eyes gives us stereo vision, allowing us to infer depth and orientation of objects. Unlike LiDAR and other sensor-based systems, this doesn't allow us to have an exact measurement for how far objects are (instead we rely on things like perspective, surface detail, and lighting conditions) or infer fine surface level imperfections, but it makes for a system that is cheap, deals well with moving objects, and is much better at dealing with complex and cohesive geometries like the ones we encounter on a daily basis. Meshes from images can be generated by using an algorithm like Structure from Motion (SfM) against multiple images from different perspectives or by using a 3D scanner such as the popular Einscan 3D



SfM — Structure from Motion with Objects (CVPR 2016) | Visual Geometry and Modelling (iit.it)

An example of a 3D model g … try — https://skfb.ly/os7RA

These models from photogrammetry are usually cheaper and lower resolution than renders from CAD drawings. However, the ease of generation often outweighs these cons.

The Many-Formats Hypothesis

One of the most frustrating things I encountered when I began working in this area was the fact that every 3D object I downloaded or came across seemed to be in a different file format. I initially chalked this up to me being inexperienced about the reasoning behind different file formats but gradually came to realize that almost every major software package for dealing with 3D models preferred to build their own format, optimized for their own software. If we assume that most 3D modeling professionals will primarily work with one piece of software (like SketchUp or Blendr) in what is often a time-consuming design and rendering pipeline, this makes sense. However, with the dizzying array of different features offered by different software packages, increasing demands in realism and complexity from 3D models, and programmatic workflows (such as the deep learning pipeline we will be building) this is highly inefficient. Images are so easy to work with primarily because there is only a handful of common formats and transferring between them is fairly straightforward. Here are just some of the formats you might encounter (caution: this may put you to sleep):

.obj: The Wavefront object format stores a tessellation of surface polygons using the position of each vertex, vertex normals, polygon faces (stored as lists of vertices), the UV position of each texture coordinate vertex, and texture vertices. Texture information is stored in a seperate .mtl file.

.stl: This is probably the most common file format for 3D printing. This is an approximate mesh format, created by tesselating the surface of an object with geometric shapes (triangles in this case). This results in a surface free of gaps or overlaps but lower surface detail then other formats. The resulting file is usually small and fairly simple to parse. Importantly, there is no color or texture information with this format.

.ply: Stanford Triangle Format, defines an object as a list of nominally flat polygons. Stored as either an ASCII or binary file and containing a generally flexible format that can be made very simple, these are usually fast to read and write. Inspired by the .obj

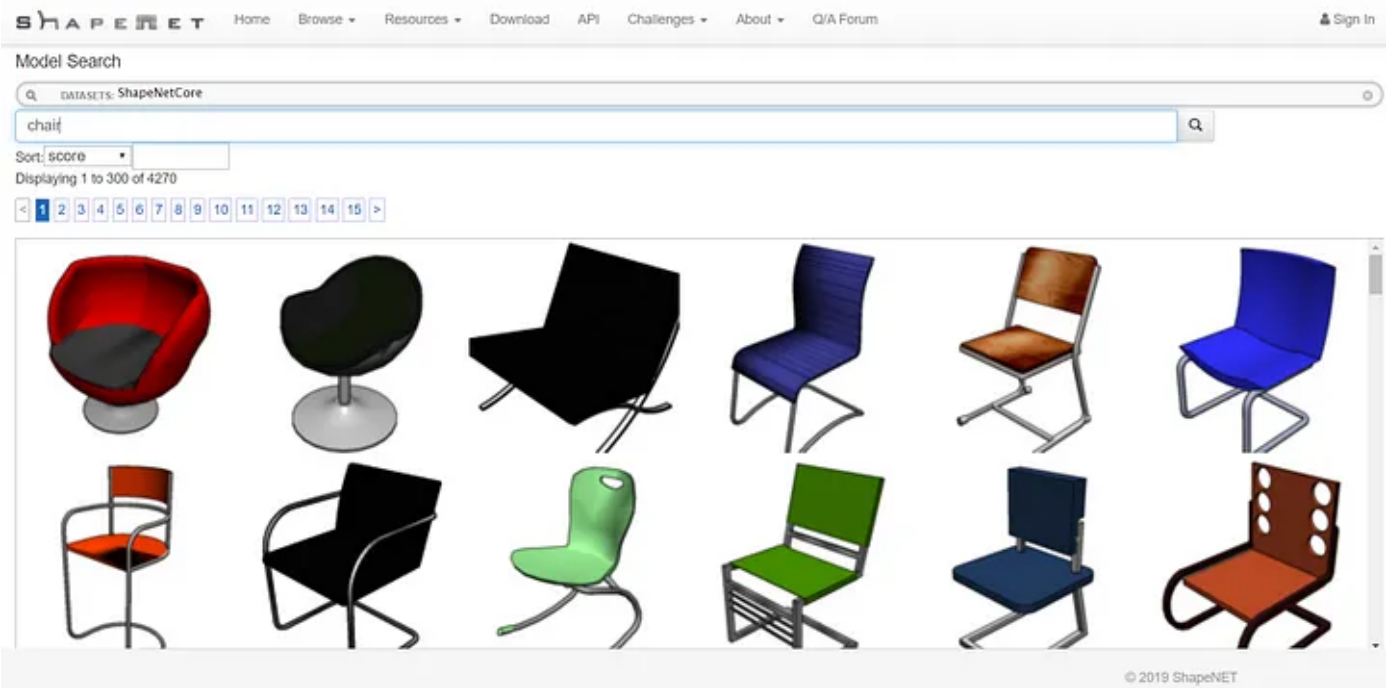
format but containing additional “property” and “element” fields, this format can encode any number of arbitrary fields relating to surface geometry (such as texture coordinates and color/transparency values) and even permits different properties for the front and back of each polygon. This is probably the most human readable format and can be manipulated easily within Python.

.dae: The collaborative design activity or COLLADA format is based on .XML and enjoyed fairly quick and widespread adoption (thanks to support by many major game studios and major 3D modeling softwares such as Autodesk). Made to be interchangable between applications without loss of information, COLLADA is moreso an intermediate format that is generally converted into a software’s proprietary format. Widely used in interactive applications, COLLADA provides support for physics, animation, and shaders. Almost all major software modeling environments (such as SketchUp, AutoCAD, 3ds Max, Autodesk, Maya etc.) support COLLADA.

Sourcing free 3D models

Many websites (https://clara.io/library, https://www.cgtrader.com/free-3d-models) host a combination of free and paid 3D model packages, however these vary in both artistic license and resolution. The best source for our purpose, feeding into a deep learning model, is Shapenet.

Shapenet spans 325 object categories and over 60,000 3D models. Best of all these are all freely available and share a consistent (and realistic) design language. The models are generated (and copyright retained) by various 3D artists using CAD drawings.



From Mesh to Point Cloud

It is infinitely easier to perform deep learning against point clouds compared to directly against mesh. This is because point clouds can easily be encoded into a (N,3) Numpy array, whereas meshes are most naturally represented by graphs. While deep learning on graphs is a fairly developed area, it is not nearly as broad as the area of deep learning methods on arrays. While the mesh has more rich surface information, the derived point cloud is much lighter in size and can be up and down-sampled as necessary. To perform the conversion, we will take a .obj model file downloaded from Shapenet, read it into Python, and perform surface sampling on it. Instead of using uniform sampling of the surface (which can lead to certain unfavorable effects), we will use the Lloyd relaxation algorithm, a great implementation of which is available at this repo. The relevant method encapsulated as a function is below:

Running this against a .obj model from Shapenet (sample available here) will result in a (n_points, 3) Numpy array, which we will save in the native .npy format.

We can download a single category of Shapenet models but note that you will need the synset id for the object, the easiest way I've found to do this for a general class of objects is to use the taxonomy browser, select an object and then click "ImageNet" on the top left. This will bring up the corresponding ImageNet page, where the ID will be contained in the URL and will be similar to "n02843684", if you remove the first "n" character this will give you the ShapeNet synsetID for the broad class.

Testing examples:
bench: 02828884
chair: 03001627
automobile: 02958343

Example URL:

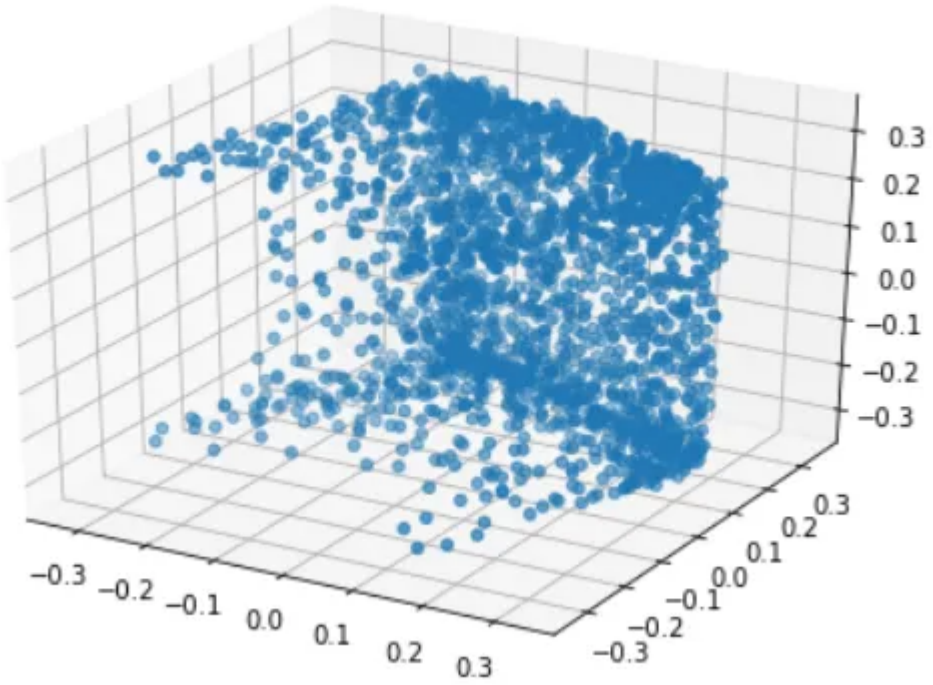
http://shapenet.cs.stanford.edu/shapenet/obj-zip/02828884.zip

After we've gone through and converted all our desired meshes into point clouds, we can visualize them using Matplotlib:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

input_pc = np.load('test.npy')
print(input_pc.shape)

x,y,z = input_pc.T
fig = plt.figure()
ax = Axes3D(fig)
ax.scatter(x,y,z)
```

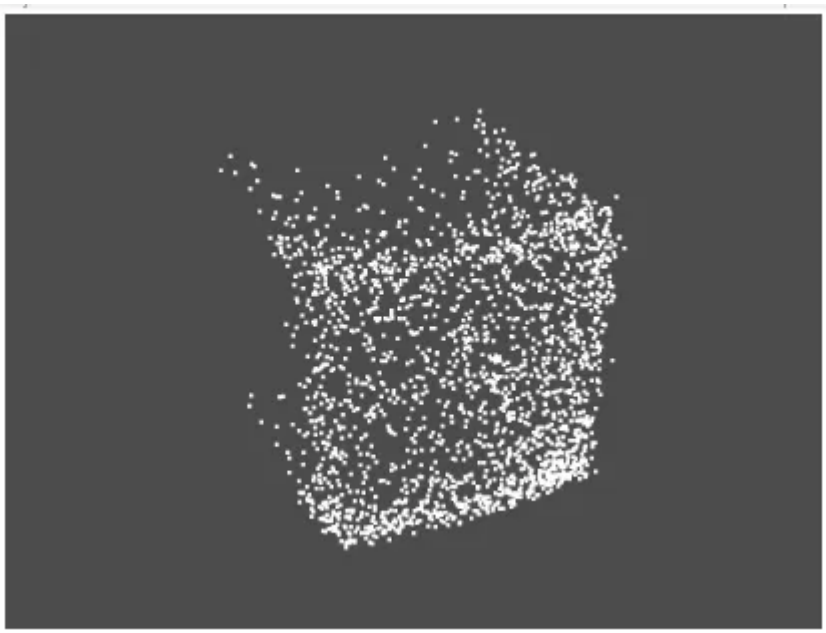


or utilizing PyVista we can get a much nicer visualization (as a bonus PyVista can also visualize meshes):

```
import numpy as np
import pyvista as pv

input_pc = np.load('test.npy')
print(input_pc.shape)

cloud = pv.PolyData(input_pc)
cloud.plot()
```



At this point, we are ready to move to part 2, where we will learn about creating a Pytorch DataLoader for our newly acquired data and about different loss functions we could use in our network. Thanks for reading!

3d Modeling Artificial Intelligence Deep Learning Metaverse