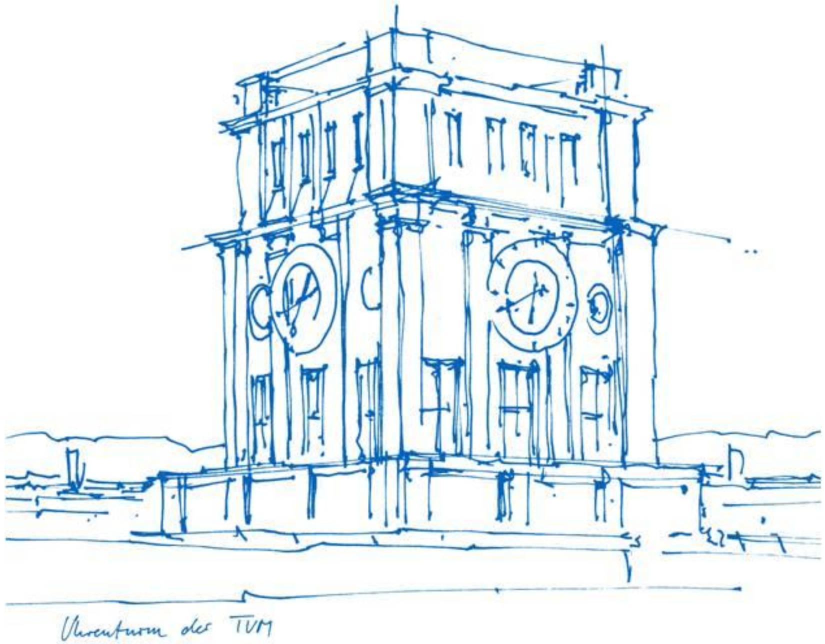
Geometrical Deep Learning on 3D Models: Classification for Additive Manufacturing

Nouhayla Bouziane | Ahmed Ebid | Aditya Sai Srinivas | Felix Bok | Johannes Kiechle

Technical University Munich

Faculty of Mathematics

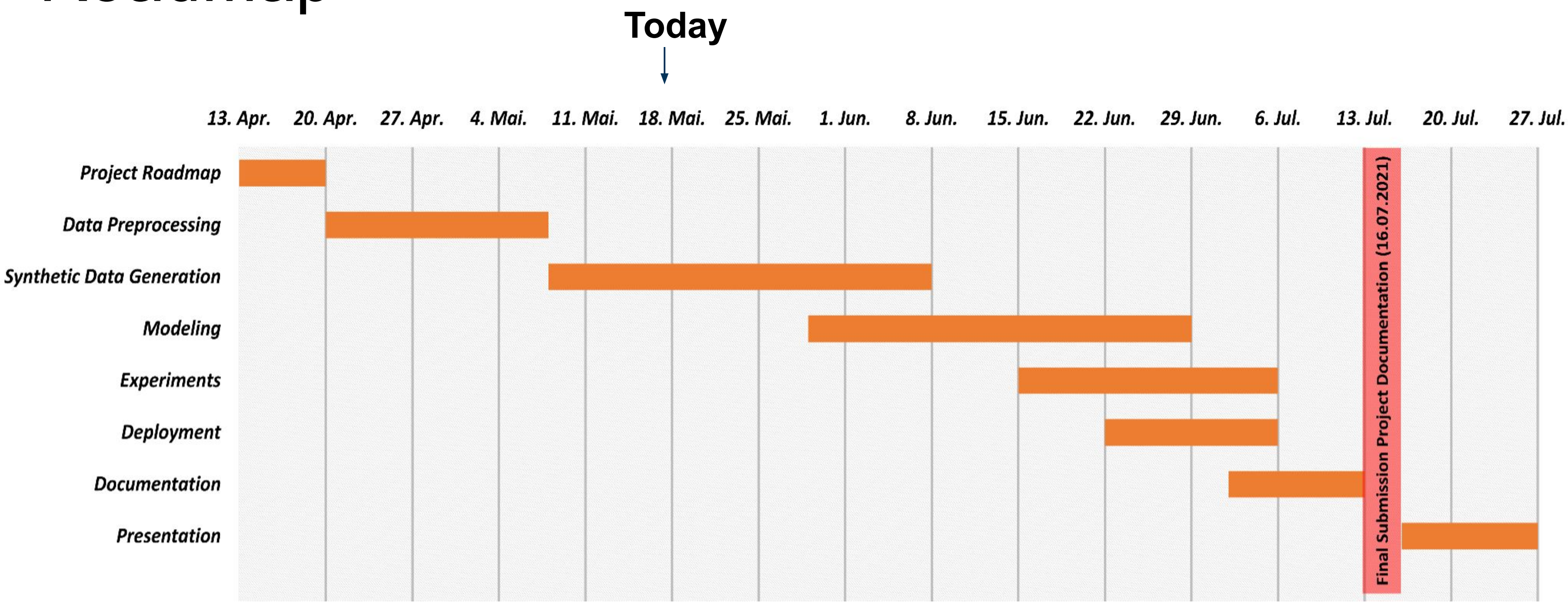
18. May 2021



Agenda

- Roadmap
- Progress:
  - Data Cleaning
  - Data Normalization and Alignment
  - Infrastructure and CUDA voxelization
  - Voxelization Algorithm and Representation
  - Visualization
  - Models Preselection
  - Basic defects insertion
- Project Structure
- Wrap Up: Next steps

RoadMap


Today
13. Apr.
20. Apr.
27. Apr.
4. Mai.
11. Mai.
18. Mai.
25. Mai.
1. Jun.
8. Jun.
15. Jun.
22. Jun.
29. Jun.
6. Jul.
13. Jul.
20. Jul.
27. Jul.
Project Roadmap
Data Preprocessing
Synthetic Data Generation
Modeling
Experiments
Deployment
Documentation
Presentation
Final Submission Project Documentation (16.07.2021)

RoadMap

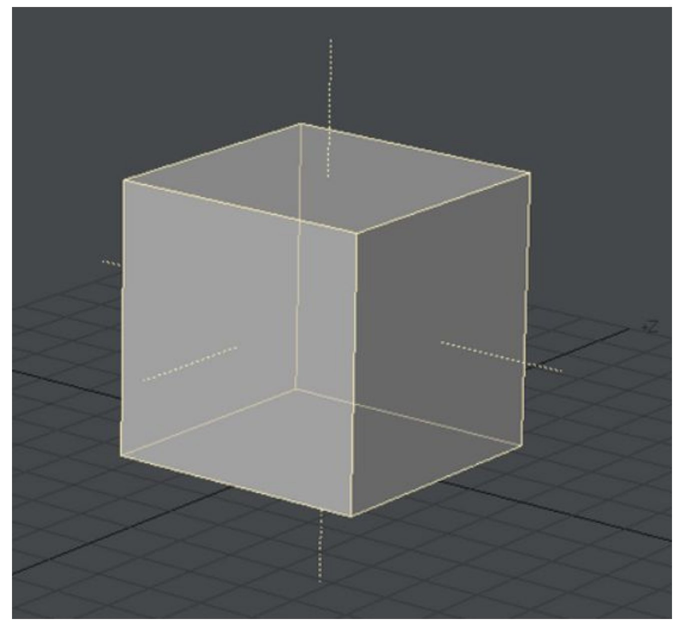
Milestones:

- Data Preprocessing
- Synthetic Data Generation ← **Currently here**
- Modeling
- Experiments
- Deployment
- Documentation

Data Cleaning

- For a 3D mesh to be 3D-printable, 2 main properties should be satisfied:
  - 1. Watertightness: mesh has no holes + normals are facing outwards

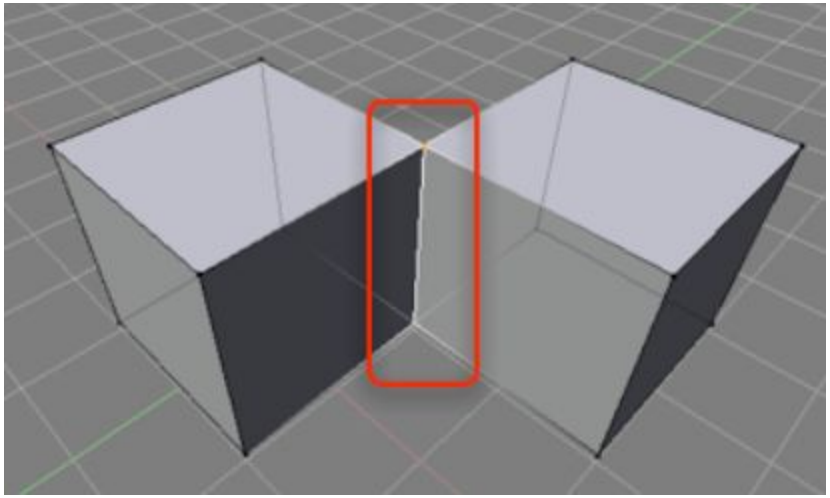
Stanford Bunny Model Bottom View
Invalid: has holes



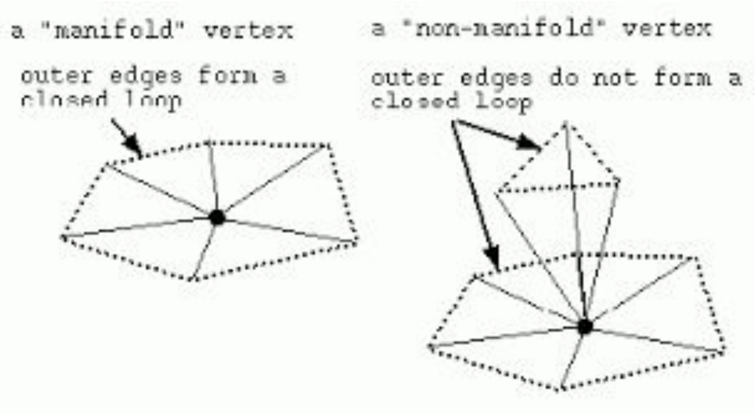
Valid: normals are facing outwards

Data Cleaning

- For a 3D mesh to be 3D-printable, 2 main properties should be satisfied
  - 2. Manifold geometry: mesh has no edges shared by more than two faces



Invalid: edge shared by 4 faces



Data Cleaning

- O3D Mesh Cleaning
  - Checks
    - Check if all vertices are manifold
    - Check if all edges are manifold
    - Check if the mesh is watertight
  - Vertices fixes
    - Remove vertices that have identical coordinates
    - Remove vertices that are not referenced in any triangle
  - Edges fixes
    - Remove non-manifold edges
  - Triangles fixes
    - Remove triangles that reference the same three vertices
    - Remove triangles that reference a single vertex multiple times in a single triangle

Data Cleaning

Using **Pymeshlab** filters to remove geometrical and topological singularities

- remove_duplicate_faces
- remove_duplicate_vertices
- remove_isolated_folded_faces_by_edge_flip
- repair_non_manifold_edges_by_removing_faces

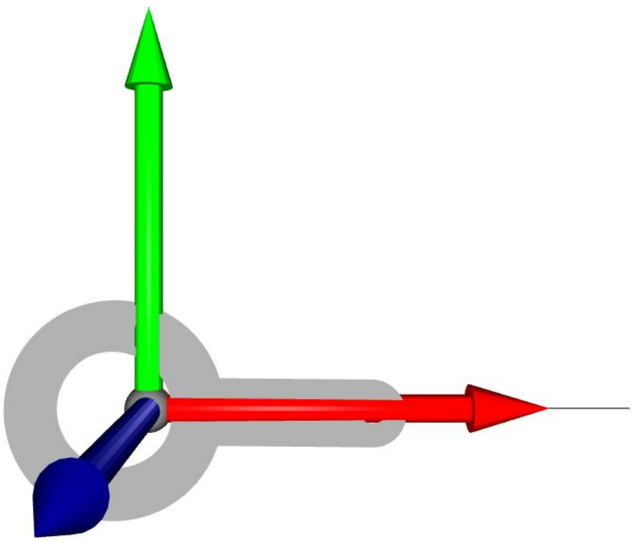
Data Normalization

- Performed to make sure all vertices of different objects lie in the same range
- Step 1: Center the mesh around the origin
  - Find the center of the mesh vertices
  - Translate the mesh vertices to the origin by subtracting the center from all vertices
- Step 2: Scale the vertices so that they lie in a [-1, 1] range
  - Divide the mesh vertices by the difference between the maximum bounding point and the minimum bounding point.

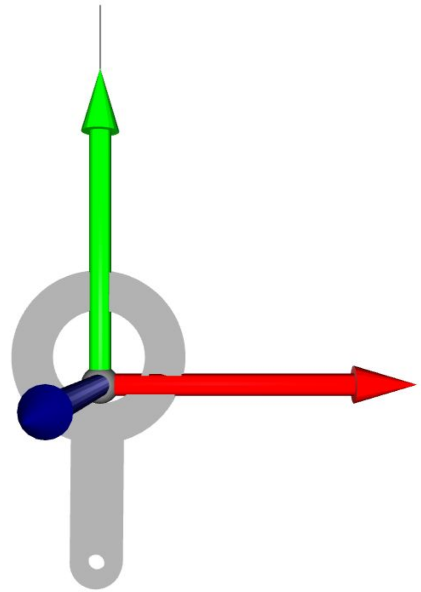
Data Alignment

- Performed to make sure all meshes are presented in the same orientation.
- The axis of the minimum Moment Of Inertia (MOI) of a mesh represents the axis around which most of the mass of the object is wrapped.
- Aligning the axis of the minimum MOI with one of the coordinate axes will allow meshes to be presented in the same orientation
- **Step 1**: Find the axis of the minimum MOI
- **Step 2**: Compute a rotation matrix that aligns the axis of the minimum MOI with a coordinate axis
- **Step 3**: Apply the rotation matrix on the mesh vertices

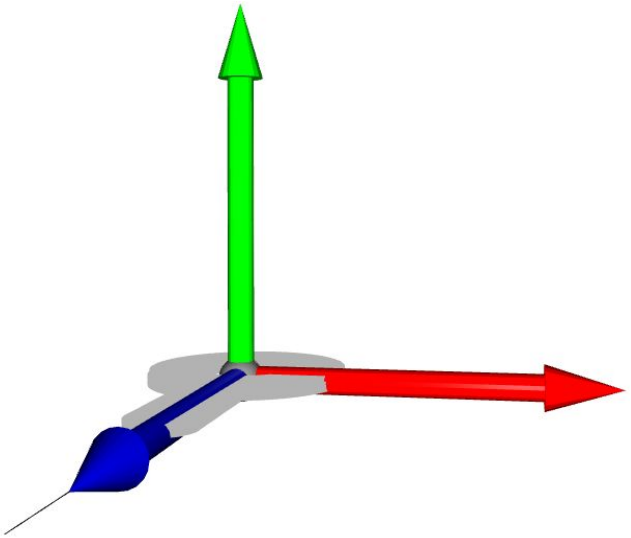
Data Alignment



Alignment of min MOI axis and X-axis



Alignment of min MOI axis and Y-axis



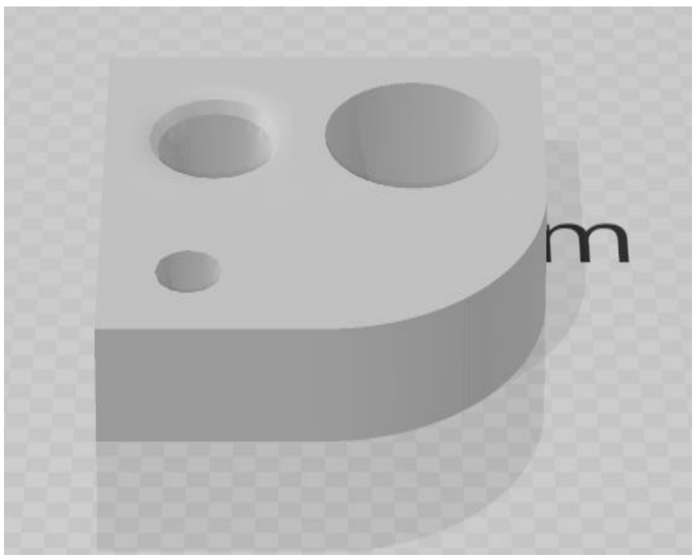
Alignment of min MOI axis and Z-axis

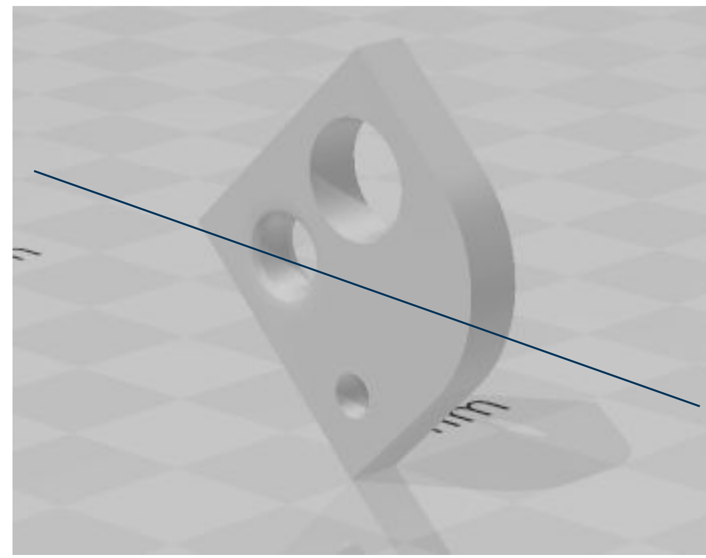
Data Alignment

Using **Pymeshlab** filter: transform_align_to_principal_axis.

- Rotating the mesh to align it with it’s principal axis of inertia.
    - The inertia matrix is a constant real symmetric matrix
    - Diagonalization allows to find a rotation that makes this matrix diagonal
    - These corresponding axes are the principal axes.

Data Alignment





Voxelization Algorithm

Voxel representations present:

- Signed Distance Function (SDF)
- Truncated Signed Distance Function (TSDF)
- Occupancy grid

Voxel representations used:

- Signed Distance Function (SDF)
- Occupancy Grid

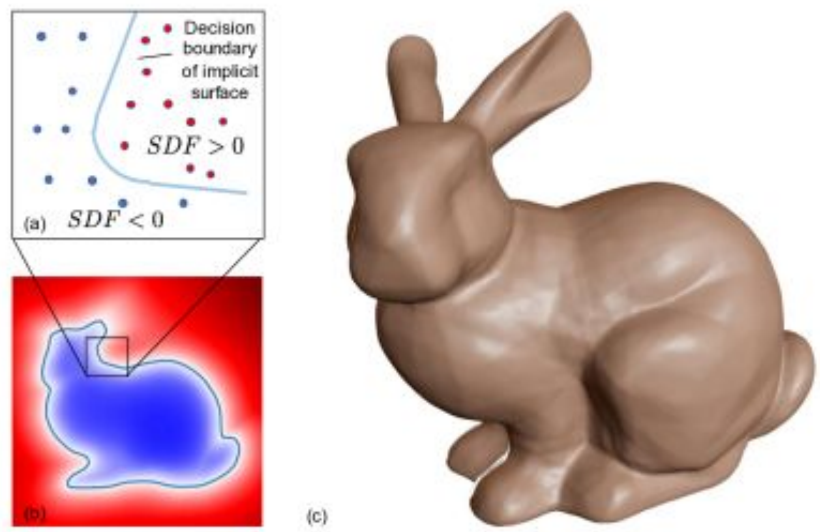
Voxelization Algorithm - SDF

- $f : R^3 \rightarrow R$

  $\{x \in R^3 \mid f(x) > 0\} \rightarrow$ Outside

  $\{x \in R^3 \mid f(x) = 0\} \rightarrow$ Surface

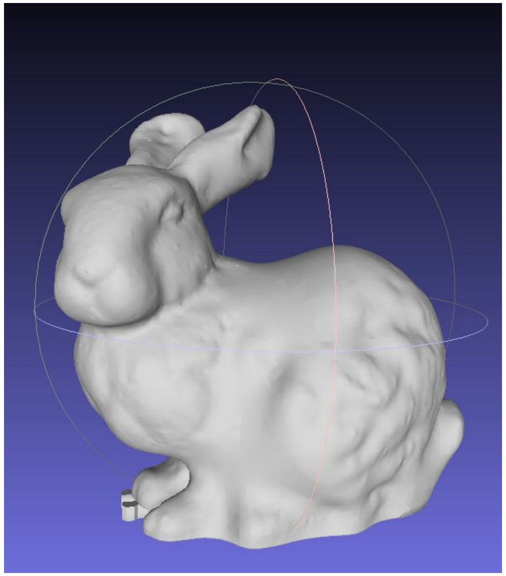
  $\{x \in R^3 \mid f(x) < 0\} \rightarrow$ Inside



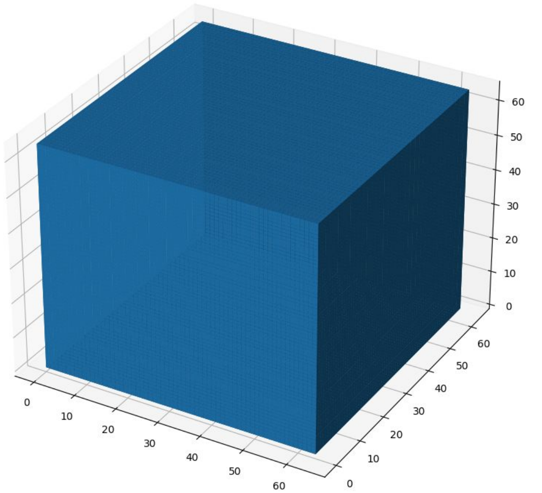
DeepSDF: Learning Continuous Signed Distance Functions for Shape Representation

- Marching Cubes algorithm is used for reconstructing the mesh and rendering.

Voxelization Algorithm - SDF



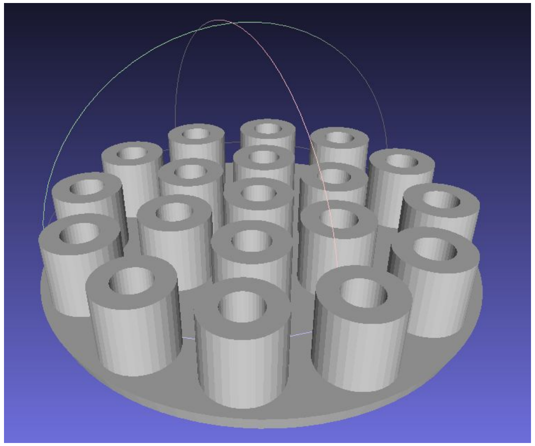
Stanford Bunny .stl file



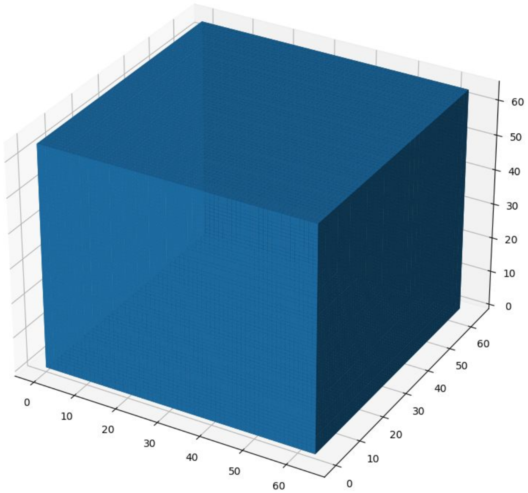
SDF representation of Stanford Bunny

Reconstructed Stanford Bunny using marching cubes algorithm

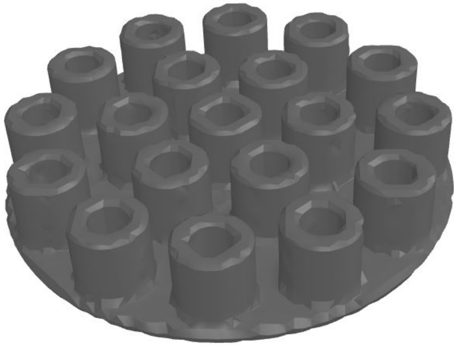
Voxelization Algorithm - SDF



.stl file from ABC Dataset



SDF representation of file from ABC Dataset

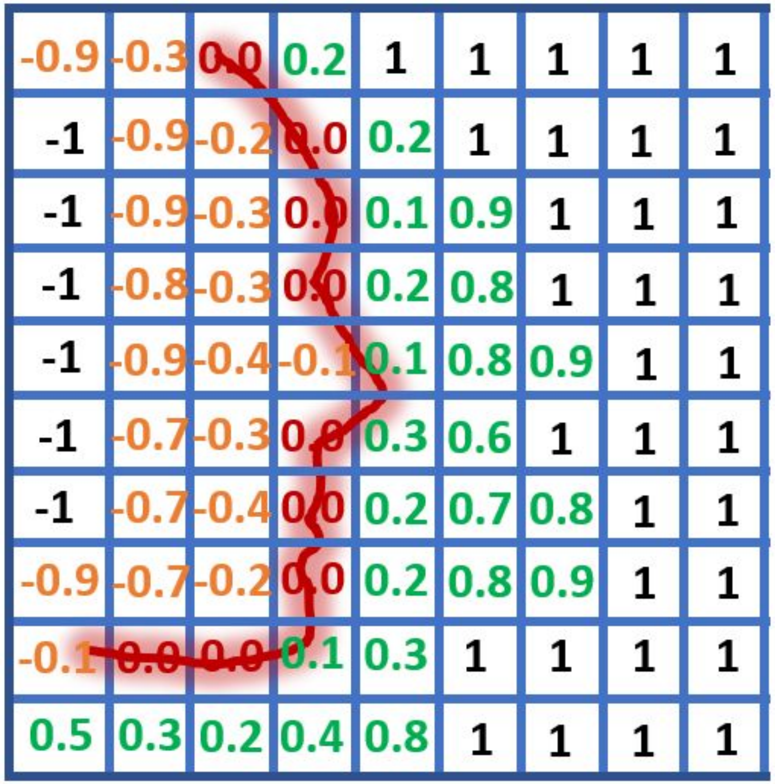


Reconstructed file from ABC Dataset using marching cubes algorithm

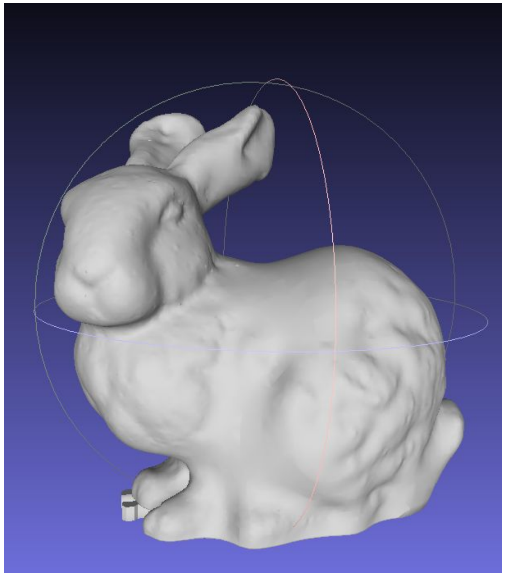
VOLKSWAGEN DATA:LAB

Voxelization Algorithm - TSDF

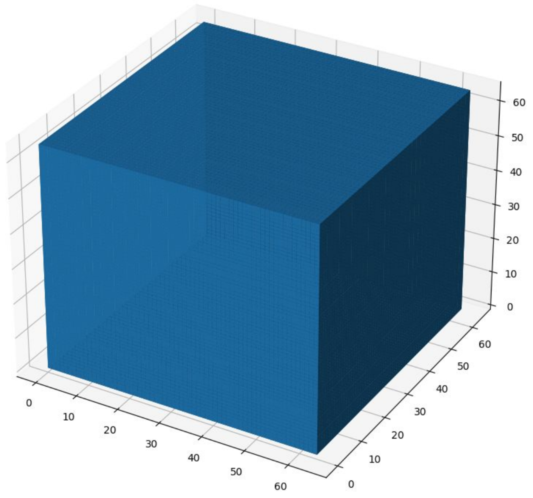
- Defines a limited SDF near the surface and truncates the value where the unsigned distance is above a specified threshold.



Voxelization Algorithm - TSDF



Stanford Bunny .stl file



SDF representation of Stanford Bunny

Reconstructed Stanford Bunny using marching cubes algorithm TSDF Representation

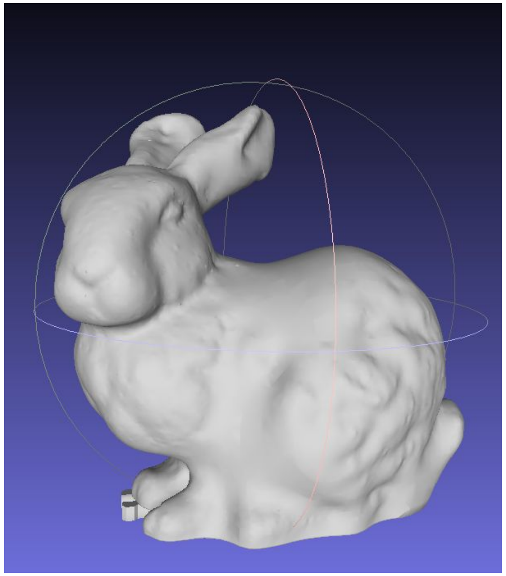
Voxelization Algorithm - Occupancy Grids

- The most straight forward voxel representation.
- Due to cubically growing compute and memory requirements, only low resolution occupancy gids (usually below 128^3) can be handled.

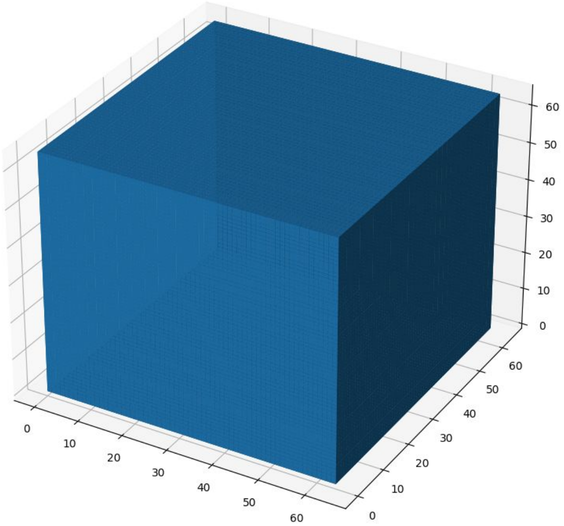
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	1	1	1	1	1	1	0	0
0	0	1	0	0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	0	1	0	0
0	0	1	1	1	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0

Source: Machine Learning for 3D geometry TUM

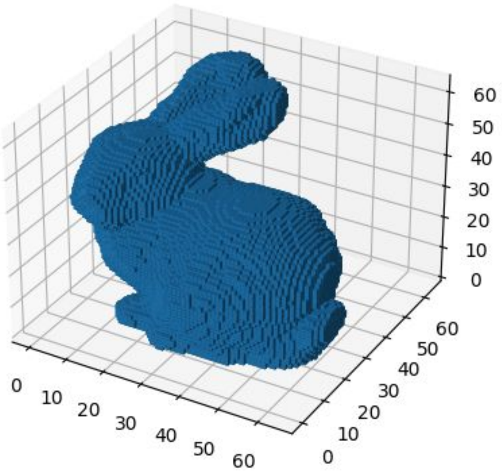
Voxelization Algorithm - Occupancy Grids



Stanford Bunny .stl file



SDF representation of Stanford Bunny



Occupancy grid representation of Stanford Bunny

VOLKSWAGEN DATA:LAB

Infrastructure @ LRZ

Available resources	Purpose
LRZ AI System	Heavy computing for neural network training
LRZ DSS	Data storage with fast link to LRZ compute resources
LRZ Compute Cloud	General purpose instance allocation

For what reason would it be instrumental to have both, AI System and Compute Cloud?

Infrastructure @ LRZ

	LRZ AI System	LRZ Compute Cloud
Permission	restricted	root access
Time limit	yes (several hours)	no
User access	one at time	several

- Permission:
  - If Ubuntu packages are missing, only LRZ Servicedesk is eligible to install on AI System, otherwise: “enroot container”
- Time limit:
  - AI System resource allocation is limited in time. At present, we do not know how computational expensive voxelization gets (depends on number of files & resolution)
- User access
  - Using a public IP and a common user account on the LRZ Compute Cloud instance would enable every group member to access computing resource

Accelerated Voxelization

What we take for granted:

- Voxelization is not feasible to be done on our local machines

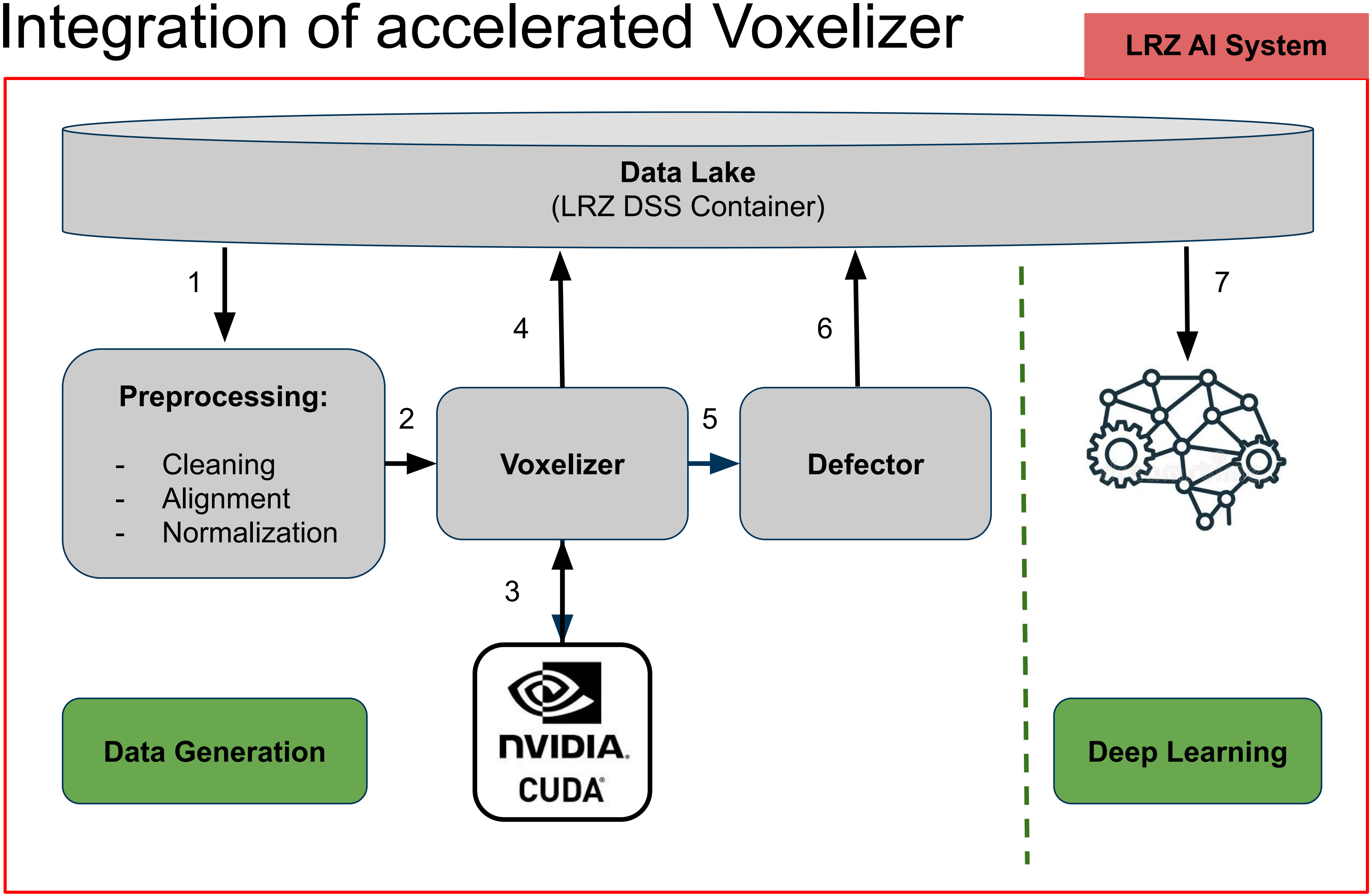
  Reason: Required resolution is supposed to be high, many 3D models to be converted

Remedy:

- Leverage optimized CPU / GPU libraries for voxelization
  - Public available GitHub repositories
    - (CPU) https://github.com/sylefeb/VoxSurf
    - (GPU) **https://github.com/kctess5/voxelizer**
    - (GPU) https://github.com/Forceflow/cuda_voxelizer

  - Nvidia GVDB Voxels
    - (GPU) https://developer.nvidia.com/gvdb (dense -> sparse)

Integration of accelerated Voxelizer



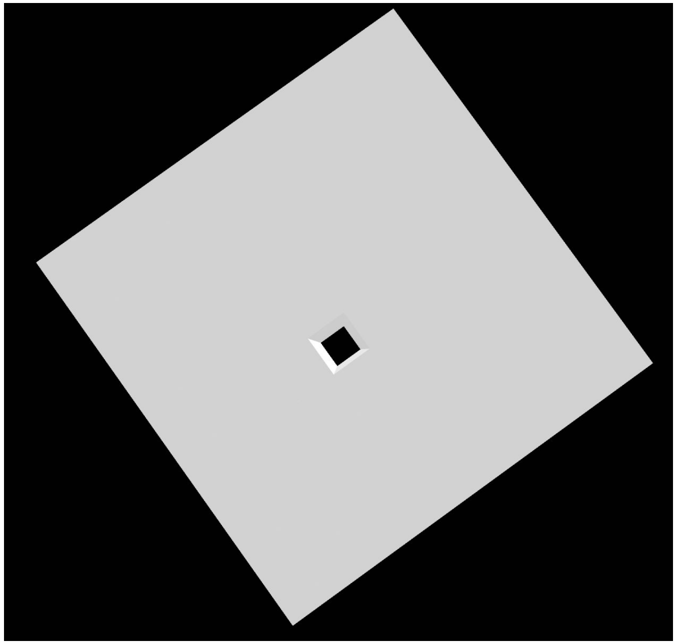
Infrastructure and Voxelization

Open-ended questions (11.05.2021):

- Required voxel resolution? -> Hyperparameter to be tuned
    - Still unknown, to be figured out
- How to integrate accelerated voxelization smoothly into data processing pipeline?
    - “VoxelizerGPU” class which invokes CUDA accelerated voxelizer + data handling
- Does the selected GPU voxelizer match with our defined interfaces?
    - Yes, output is converted into compressed numpy array format (binvox -> npz)
- Which voxelization technique is used and does this match our desires?
    - Occupancy grid, dense representation, sparse representation (possible)

Adding basic defects

Removing one stack of voxels

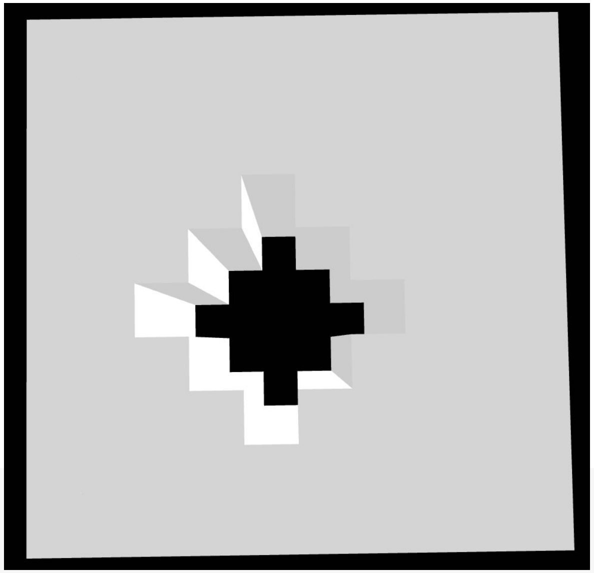


```
voxel_sim = np.ones((50, 50, 50), dtype=np.int32)
voxel_sim_defect = deepcopy(voxel_sim)
voxel_sim_defect[:, 5, 5] = 0
vis.plot_voxel(voxel_sim_defect, 'stack_removed', save=False)
```

VOLKSWAGEN DATA:LAB

Adding basic defects

Removing basic hole

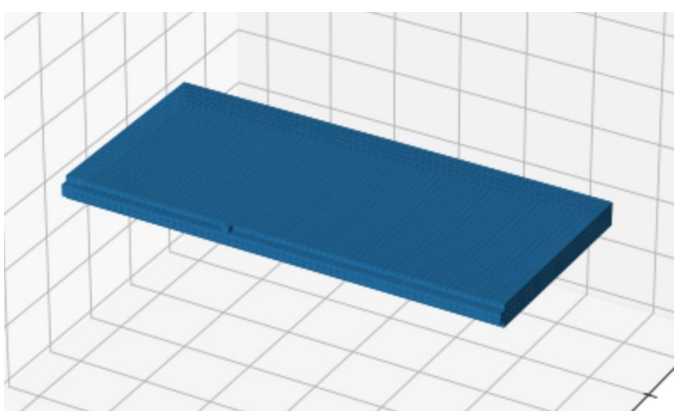


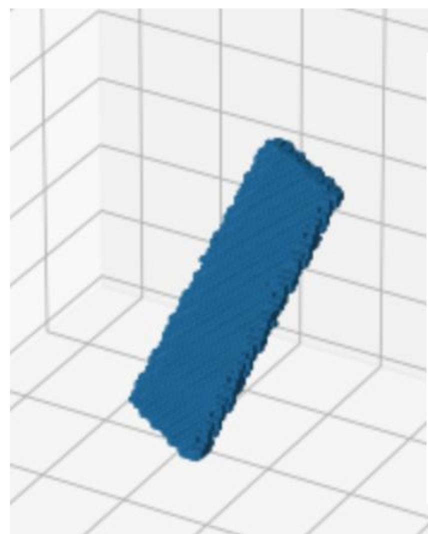
```python
radius = 2
voxel_sim_defect_circle = deepcopy(voxel_sim)
xx = np.arange(voxel_sim.shape[0])
yy = np.arange(voxel_sim.shape[1])
out = []

for idz in range(voxel_sim.shape[2]):
    voxel_sim_defect_circle_cut = voxel_sim_defect_circle[5, 5, idz]
    inside = (xx[:,None] - 4) ** 2 + (yy[None, :] - 4) ** 2 > (radius ** 2)
    out.append(voxel_sim_defect_circle_cut & inside)

vis.plot_voxel(np.array(out), 'test_big_hole', save=False)
```

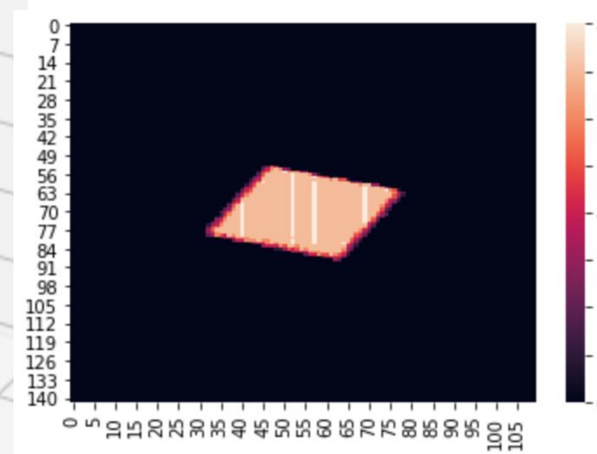
Adding Basic Defects

Going generic

1. Rotate the voxelized model randomly around each axis via rotation matrix and interpolation
2. Analyse the top-view and determine the possible place to add a vertical hole
3. Out of the before defined subset, randomly choose one place to add a vertical hole
4. Rotate the model back
5. Check for artefacts (f.e. floating voxels) - open task



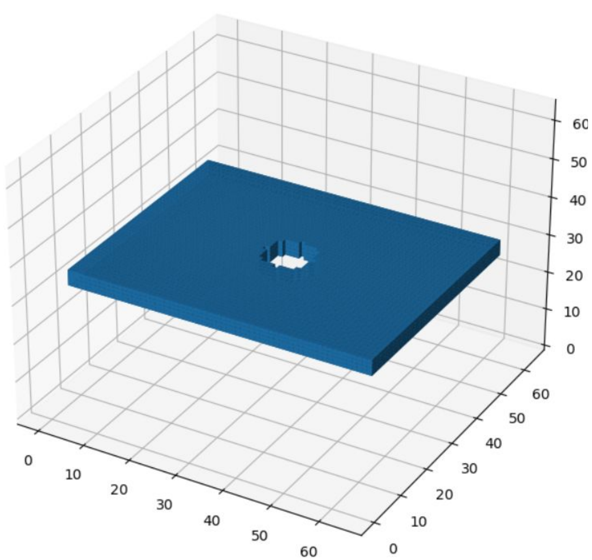


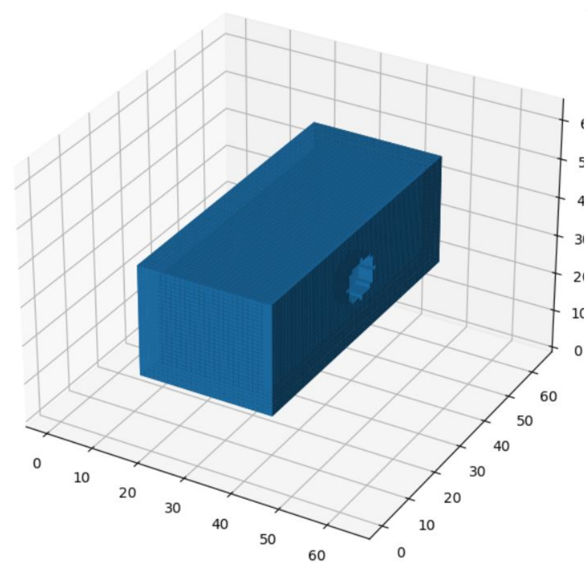


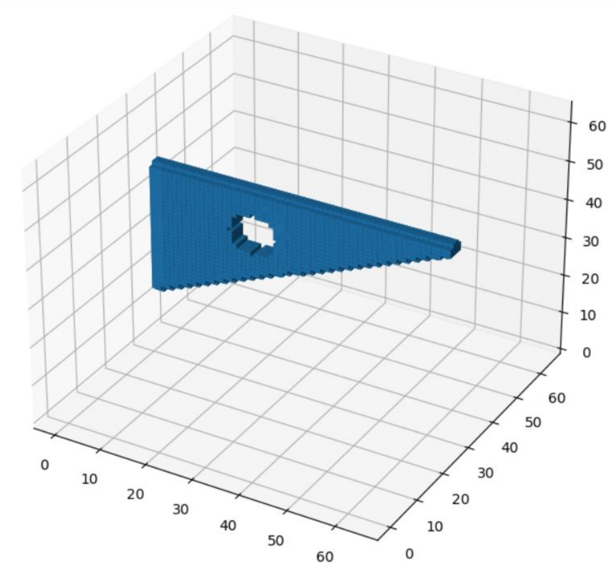
Adding Basic Defects

Automated defects Adder

Greedy approach to find an appropriate orientation and size for defect insertion



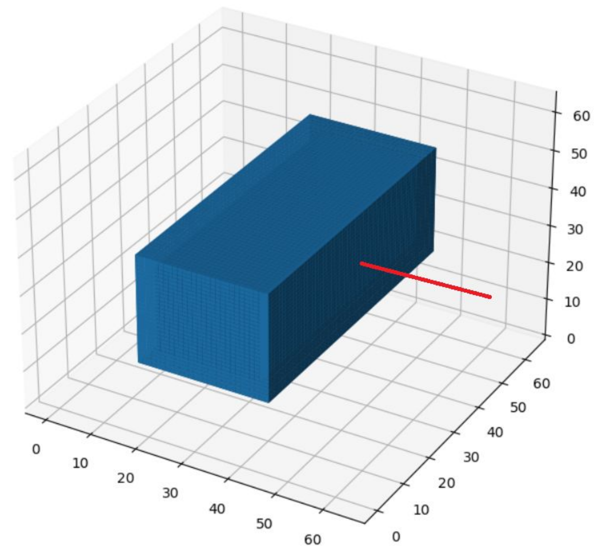




Adding Basic Defects

Automated defects Adder

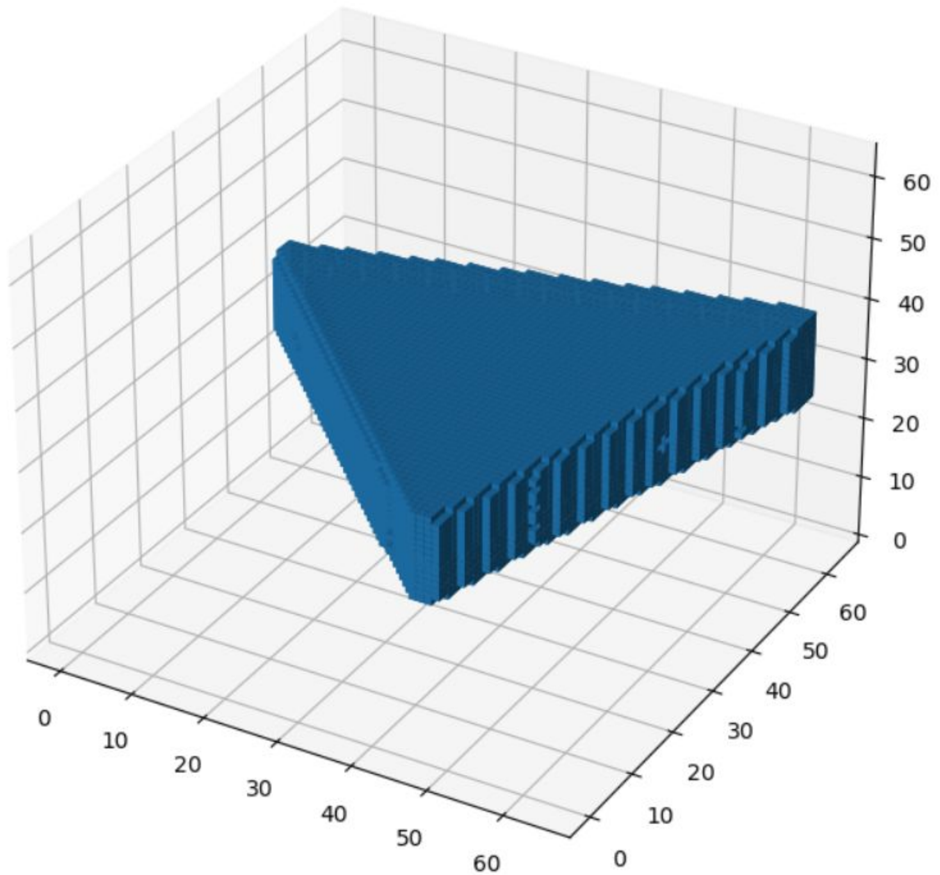
- Step #1: Given an occupancy grid of 64 voxels, find the grid indices
- Step #2: Find an appropriate axis and size for the hole
  - Loop through the axes (X, Y, Z)
  - For the chosen axis, get the minimum dimension of the perpendicular plane
  - Loop through a set of predefined hole diameters [10 … 2]
  - Find the difference between the min dimension and the hole diameter
  - If the difference is greater than 10, stop
- Step #3: Define the cylinder and find the voxels inside
- Step #4: Remove the voxels



Adding Basic Defects

Automated defects Adder

Drawback: greedy, not optimal



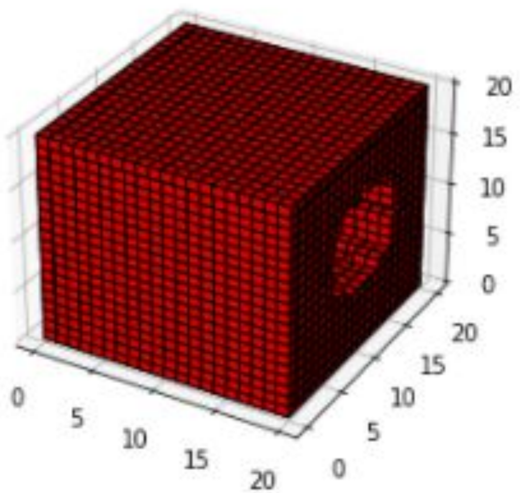
Adding Basic Defects

Removing rotated holes

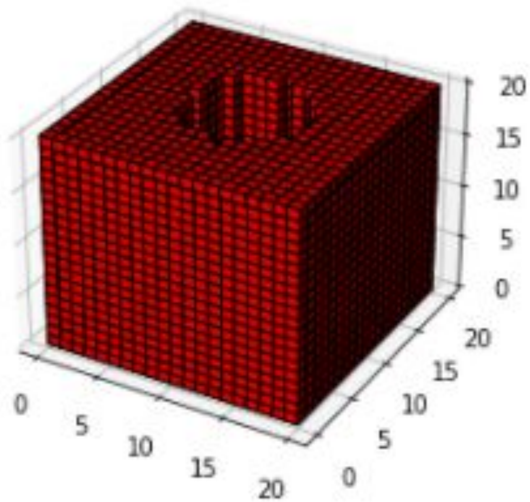
1. Create a 4d grid with the indices of the voxels using the shape of our model
2. Set the radius and height of the cylinder
3. Get the voxels inside a cylinder rotated with an angle defined
4. Remove the voxels inside the cylinder from our model

Adding Basic Defects

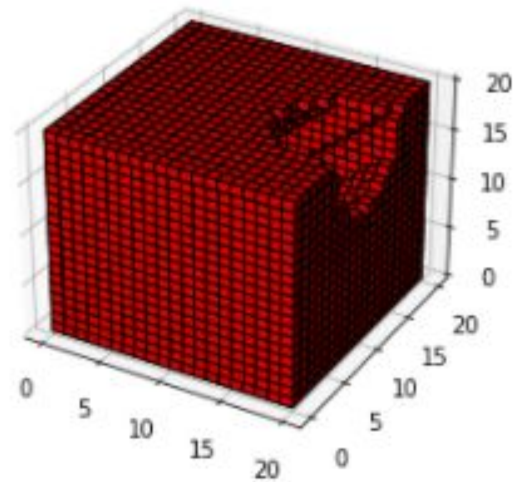
Removing rotated holes



- Rotation in (x,z) direction
- Angle = 90
- Radius = 5



- Rotation in (x,z) direction
- Angle = 180 deg
- Radius = 5

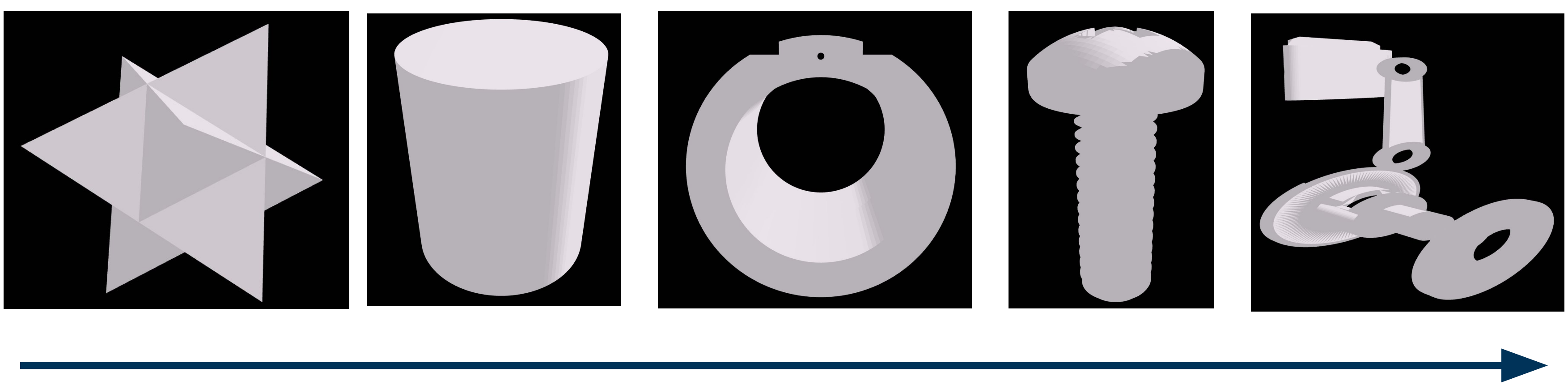


- Rotation in (x,z) direction
- Angle = 45 deg
- Radius = 5

Visualization

Solution implemented:

- Visualizer class with methods to plot meshes and voxels
- Usage of plotly
    - Benefits: Easy to handle and to extend, interactive JavaScript plot, can be saved as a html file
    - Cons: RAM resources, plot cannot load



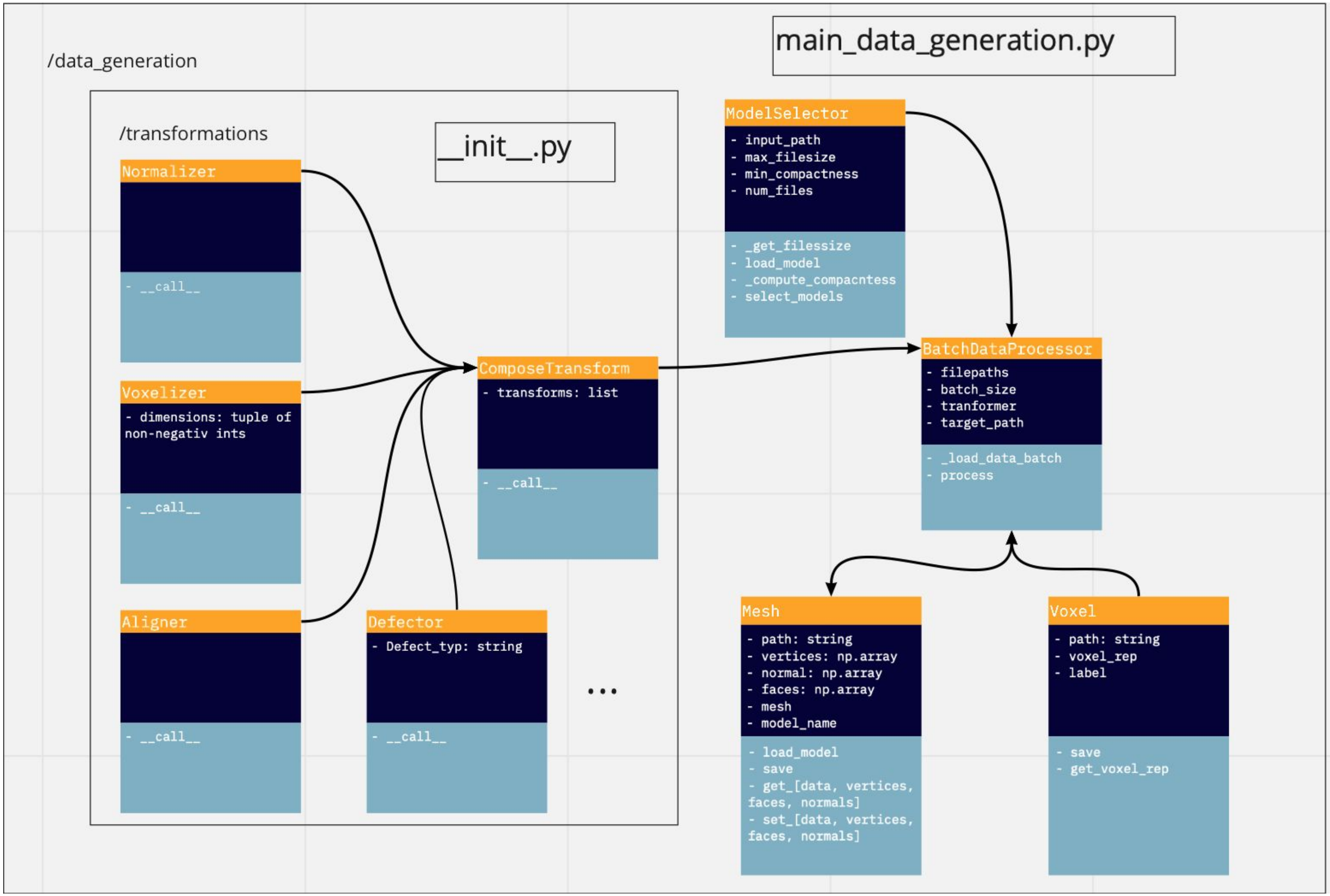
Increasing file size

Visualization

Plotting voxel models:

1. Remove voxel inside the model
   a. Calculate the number of adjacent voxels with convolution
   b. Identify voxels that have 6 neighbors
   c. Overlay filter with the model and define voxels to remove
2. Identify indices on where to plot cubes
3. Loop over each position
   a. Define all vertices of the cube
   b. Add cube to the plot

Project Structure



Wrap Up: Next Steps

- Deep learning (in progress)
- Collaboration on automated defector
- Define data labeling ruleset