LiDAR-DenseSeg: A Framework for Aerial LiDAR Semantic Segmentation, Densification, and Planar Flattening for Improved Voxelization and Mesh Reconstruction

Ashtan Mistal
University of British Columbia

Abstract

This paper introduces LiDAR-DenseSeg, a novel framework designed to enhance voxelization and mesh reconstruction of airborne LiDAR data. The process involves three pivotal steps: Semantic segmentation, densification, and planar flattening of the point cloud. Using a modified PointNet++ architecture, the framework effectively segments point cloud data, focusing primarily on building structures and achieving an evaluation accuracy of 94.1% and a mean IoU of 0.89. Following segmentation, a recursive median split algorithm based on the SAPCU architecture densifies the point cloud, addressing the inherent sparsity in airborne LiDAR data. Planar flattening is proposed to further refine the process, reducing noise and enhancing voxelization quality. The paper presents empirical results demonstrating significant improvements in the voxelization and mesh reconstruction of airborne LiDAR data, contributing to the field of LiDAR data processing and 3D reconstruction. Code is available at https://github.com/ashtanmistal/LiDAR-DenseSeg.

1. Introduction

Voxelization and mesh reconstruction of airborne LiDAR data inherently suffers from numerous issues, including but not limited to: (1) the lack of complete semantic labels, (2) the sparsity of the data, and (3) the irregularity of the point cloud. A naive approach of voxelization leads to poor reconstruction quality and an inability to capture the fine details of the point cloud [1]. Lack of complete labels leads to the direct voxelization not being a complete representation of the available point cloud data, and entire sections of the point cloud being lost in the voxelization process due to a lack of labels. The irregularity of the point cloud leads to neighbouring points that lie along a plane in the real world to be mapped to different voxels, leading to massive amplification of noise if the voxelization is performed naively. The sparsity of the data leads to the voxelization and mesh reconstruction processes being unable to capture the fine details of the point cloud. This paper proposes a framework for improving the voxelization and mesh reconstruction of airborne LiDAR data by first semantically segmenting the point cloud, then densifying the point cloud, and finally performing planar flattening on the densified point cloud.

The semantic segmentation of the point cloud is performed using a PointNet++ architecture [2] that is trained on the already labelled data. The data is a partially labelled point cloud of the University of British Columbia campus, with the main labels of buildings, trees, water, and ground. This framework focuses on labelling buildings, as these are the most important features to capture in the voxelization and mesh reconstruction process and the ones that suffer the most from the aforementioned issues. The semantic segmentation problem is thus formulated as a binary problem, with the two classes being *buildings* and *not buildings*.

The densification of the point cloud is performed using a recursive median split implementation of the SAPCU architecture [3] using pre-trained weights from the authors. A weighted nearest neighbours algorithm is used to transfer the colour information from the original point cloud to the densified point cloud. The densification task is performed only on the buildings, and primarily enhances mesh reconstruction of the buildings given the large voxel size in the voxelization process.

To resolve the issue of the irregularity of the point cloud and fix the amplification of noise that occurs during the voxelization process, planar flattening is performed on the densified point cloud. For points that lie on a plane, they are brought to the plane along their normal. As a result, points are much more likely to get voxelized to the same voxel as their neighbours. This leads to a massive reduction in the noise of the point cloud, and an increase in voxelization quality.

We present the following contributions:

- A framework for improving the voxelization and mesh reconstruction of airborne LiDAR data.

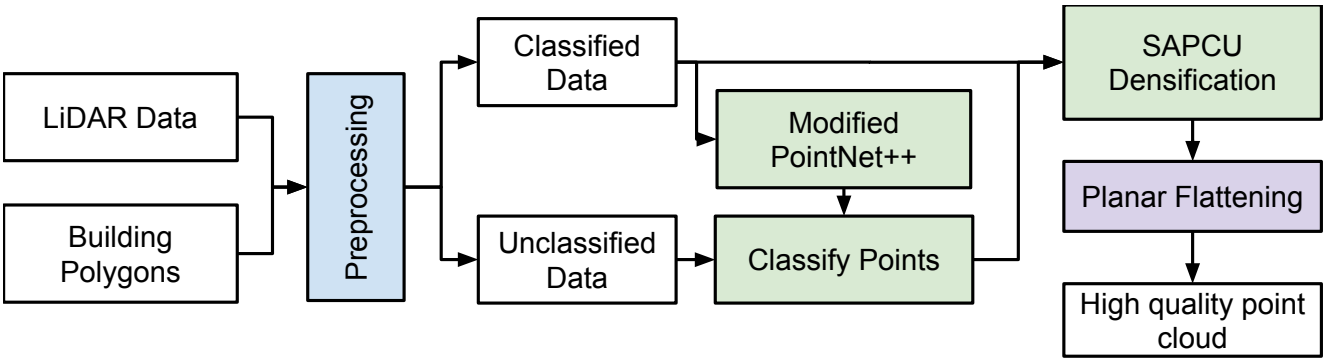

Figure 1. The LiDAR-DenseSeg pipeline. Data streams are in white, data processing is in blue, deep learning tasks are in green, and mesh processing is in purple. The high quality point cloud is the output of the pipeline.

- A trained PointNet++ architecture for semantically segmenting buildings in airborne LiDAR data.
- A recursive median split implementation of the SAPCU architecture for densifying arbitrary-size and arbitrary-scale point clouds.
- Implementation of planar flattening for reducing noise in the voxelization process.
- Empirical results showing the effectiveness of the proposed framework in improving the input point cloud for voxelization and mesh reconstruction.

2. Related Work

MinecraftUBC. LiDAR-DenseSeg is built on top of the MinecraftUBC [1] project, which is a proof-of-concept for voxelization of airborne LiDAR data into a Minecraft world. The voxelization was done in a layered approach. The first layer was surface reconstruction of the ground data, which was done using a flood-fill algorithm. This was the colorized using a pre-existing GeoJSON dataset denoting ground information. The buildings were then placed using direct voxelization of the building data using truncation to the nearest voxel. Additional GeoJSON data was used to place trails, traffic impactors, and other similar features. Trees were then placed using a combination of direct voxelization and mean shift clustering for trunk placement. This work utilizes the same dataset as MinecraftUBC, and builds on top of the naive building reconstruction algorithm used in MinecraftUBC. Ultimately, the building reconstruction algorithm used in MinecraftUBC suffered from numerous problems, including but not limited to: (1) the lack of complete semantic labels, (2) the sparsity of the data, and (3) the irregularity of the point cloud. This meant that quality buildings had to be manually edited and colorized, and the voxelization process was unable to capture the fine details of the point cloud. It is thus used as a baseline for comparison in the evaluation of LiDAR-DenseSeg.

PointNet++ Architecture. The PointNet++ architecture [2] is a deep learning architecture for point cloud classification and segmentation that is built on top of the PointNet architecture. It utilizes local point features and global point features to classify and segment point clouds. Pre-trained (and readily available) models in PyTorch related to building semantic segmentation are few and far between, and so a PointNet++ architecture is trained on the available building data of the UBC campus.

SAPCU Architecture. The SAPCU architecture [3] is a deep learning architecture for self-supervised arbitrary-scale point cloud upsampling that utilizes an implicit neural representation to upsample point clouds. The model is trained on a normalized watertight subset of ShapeNet [4,5] and based off of Occupancy Networks [5] and DGCNN [6]. Though the model is trained on watertight point clouds, it is able to generalize to other point clouds including non-watertight point clouds [3]. A drawback of the model is that it is unable to perform effectively on large point clouds due to implementation limitations. Additionally, the model is unable to transfer colour information from the original point cloud to the densified point cloud.

3. Methodology

Given a partially labelled point cloud and 2d polygons for building footprints, LiDAR-DenseSeg reconstructs buildings in the input aerial LiDAR point cloud through a three-step process: (1) semantic segmentation, (2) densification, and (3) planar flattening. The pipeline outlined in Figure 1 is used to perform this task.

3.1. Preprocessing

The original LiDAR data comes separated as square kilometre tiles. This sometimes leads to buildings being split across multiple tiles, and so the tiles are merged together to form a single point cloud. This single point cloud is then filtered into points that lie within a 5m radius of the

Figure 2. The naive voxelization of the point cloud. The lack of complete labels leads to the voxelization not being a complete representation of the available point cloud data, and entire sections of the point cloud being lost in the voxelization process due to a lack of labels. Here, every single wall in the Pharmeceutical Sciences building is not captured in the voxelization.

building footprints, reducing the dataset size from 40 GB to 3.8 GB. These footprints are gathered from a GeoJSON dataset [7] that contains the building footprints of the University of British Columbia campus, along with corresponding information regarding their name, construction status, height, and other similar information. Data related to under-construction buildings are not considered for the purposes of this project, and so the corresponding footprints and related points are removed from the dataset. Lastly, the color data associated with the dataset is quantized to the 0-255 range (from the 0-65535 range) to reduce size, and each building is saved as a separate file in the .npy format after the final data processing tasks performed in the dataloaders.

3.2. Semantic Segmentation

The points already labelled as buildings in the input point cloud do not cover significant regions of some buildings, and the quality of the reconstruction is significantly hindered by the lack of complete labels as seen in Figure 2. This data is present in the dataset, but is not labelled. The semantic segmentation task is to train a model to segment points belonging to buildings, and apply the model to the unclassified data in the point cloud. This is done using a modified PointNet++ architecture [2] that is trained on the labelled data. The modifications and additions made in the PointNet++ architecture are as follows:

KNN Fallback in Ball Query. The ball query in the PointNet++ architecture is modified to fallback to a KNN query if the number of points in the ball query is less than a threshold. This is due to the extreme sparsity in some areas of the data, where the ball query is unable to find enough existing points within the ball to extrapolate the local features. Selecting less points during the training process mitigates this issue, but lower number of points leads to a lower quality of the classification. As a result this fallback ensures the model can still perform well in these areas by extending the context as needed.

DataLoader. A dataloader is implemented to load the data from the dataset and perform data augmentation. To ensure ease of use with other datasets and avoid manual overhead, the dataloader implements some later preprocessing tasks of the data. Buildings that are underground or otherwise have a total height of less than 3m are removed from the dataset. The dataset is then split into points that are already classified (used for training and testing) and points that are unclassified (where we apply the learned model). Lastly, the LiDAR data is by default in the EPSG 26910 coordinate system; we offset the data to be zeroed on the UBC campus, and apply a rotation matrix that aligns the coordinate system with UBC's roads instead of true north. These are the same transformations that were applied in the MinecraftUBC project [1].

Upon initialization, the dataloader loads the data from the dataset and performs data augmentation. The label weights are calculated using equation 1.

$$w_j = \frac{(\max(f)/f_j)^{\frac{1}{3}}}{\sum_{k=1}^{n}(\max(f)/f_k)^{\frac{1}{3}}}, \quad j = 1, \ldots, n \qquad (1)$$

Here, w_j is the weight for class j, f_j is the relative frequency of class j, $\max(f)$ is the maximum relative frequency among all classes, and n is the number of classes. Given weights are not able to be calculated when using the model on unlabelled data, the weights are calculated using the labelled data and then passed to the model when using the model on unlabelled data.

Training. The model is trained on the labelled data using the Adam optimizer with an initial learning rate of 0.01, a batch size of 32, a weight decay of 0.0001, and a learning rate scheduler that reduces the learning rate by a factor of 0.7 every 10 epochs. Each building is passed individually to the model, and the centroid is calculated through a random sampling that ensures sufficient points in the block. The block is then centered on the centroid in the x and y axes, and the minimum height of the block is set to the minimum height of the building. The data is normalized, and both the normalized and non-normalized data is passed to the model.

With the exception of the KNN fallback in the ball query, the rest of the PointNet++ architecture is left unchanged and used mostly as a black box. The model is trained on the labelled data, and then applied to the unclassified data to segment the buildings.

A block size of 32.0m is used for the semantic segmentation, which aims to capture sufficient points in the block while ensuring applicability to smaller buildings without relying on the KNN fallback. 4096 points are sampled from each block. The training is split into 70% training data and 30% evaluation data, and the model is trained for 64 epochs.

Figure 3. The Walter Gage residence reconstructed using the naive voxelization algorithm used in MinecraftUBC. Note the lack of detail in the deck on a per-floor basis caused by the sparsity of the data.

3.3. Densification

The classified point cloud is then improved further through densification. The model used for densification is based on SAPCU [3], which is a deep learning architecture for self-supervised arbitrary-scale point cloud upsampling that utilizes an implicit neural representation to upsample point clouds. Upsampling point clouds is performed with the goal of solving the problem of sparse point clouds, which is inherent in airborne LiDAR data. Voxelization without this task performed leads to poor reconstruction quality and an inability to capture the fine details of the point cloud, as seen in Figure 3.

In order to upsample the point cloud, we aim to address some of the limitations of the SAPCU architecture. Most notably, the SAPCU architecture is unable to perform effectively on large point clouds due to implementation limitations. Simply increasing the size of the KDTree used in the model does not solve this issue, as the size remains fixed and unable to adapt to variable point cloud sizes. Additionally, increasing this size leads to errors in the model with a lower number of points. As a result, the model is unable to upsample the entire point cloud at once. We thus implement a recursive median split algorithm that splits the point cloud into smaller chunks, densifies each chunk, and then merges the chunks back together.

The split is based on the median of the point cloud along the largest dimension, which is determined by the bounding box of the point cloud. The recursive algorithm is called with each split until the number of points in the point cloud is less than a threshold, which is 5000 as per the original implementation of the SAPCU architecture. In the base case of less than 5000 points, the data is normalized and passed to the model, and then re-scaled to the original size. The pseudo-code for the recursive median split algorithm is shown in Algorithm 1.

The preprocessing tasks performed in this algorithm are simply that of handling the .obj files that the semantic segmentation task outputs, and selecting only the points that are classified as buildings.

Algorithm 1: Recursive Median Split Algorithm

```
Function recursiveMedianSplit(pointCloud):
    if len(pointCloud) < 100 then
        return pointCloud
    else if len(pointCloud) < 5000 then
        pcN ← normalize(pointCloud)
        pcN ← SAPCU(pcN)
        pc ← rescale(pcN)
        return pc
    else
        bbox ← getBoundingBox(pointCloud)
        splitDimension ←
          getLargestDimension(bbox)
        median ← getMedian(pointCloud,
          splitDimension)
        left, right ← split(pointCloud, median,
          splitDimension)
        left ← recursiveMedianSplit(left)
        right ←
          recursiveMedianSplit(right)
        return merge(left, right)
```

3.4. Planar Flattening

In order to resolve the issue of the irregularity of the point cloud and fix the amplification of noise that occurs during the voxelization process, planar flattening is performed on the densified point cloud. For points that lie on a plane, they are brought to the plane along their normal. To determine what points lie on a plane, a plane detection algorithm is used. A point is selected at random from the unseen points, and a plane is fit to the point and its k nearest neighbours. This plane fit is performed using a 3D RANSAC algorithm. If the plane is a good fit, then k is increased and the plane is fit again using these new points. Whether or not a plane is a good fit is determined by the number of points that lie within a threshold distance of the plane. If the plane is not a good fit, the next smallest plane that was a good fit is selected and the points marked as seen. This process is repeated until all points are seen. Once all points are seen, the points that lie on a plane are brought to the plane along their normal.

The pseudo-code for the plane detection algorithm is shown in Algorithm 2.

Once all points are fit to a plane, only planes with n points or greater are considered; smaller planes are either corners, edges, or other non-planar features. The points that lie on these planes are then brought to the plane along their

Algorithm 2: Plane Detection Algorithm

```
Function planeDetection(pointCloud, threshold):
    unseen ← pointCloud
    planes ← []
    while len(unseen) > 0 do
        point ← randomChoice(unseen)
        k ← 3
        initNeighbours ← getNeighbours(point, k)
        prevPlane ← fitPlane(initNeighbours)
        repeat
            neighbours ← getNeighbours(point, k)
            plane ← fitPlane(neighbours)
            if len(plane) > threshold then
                k ← k + 1
                prevPlane ← plane
            else
                append(planes, prevPlane)
                remove(unseen, prevPlane.points)
        until length of plane ≤ threshold
    return planes
```

normal. This is done by calculating the distance from the point to the plane, and then moving the point along the normal by that distance.

4. Results

4.1. Semantic Segmentation

Quantitative Results. After 64 epochs, the model achieves an evaluation accuracy of 94.1%, with a mean IoU of 0.89. The IoU for non-buildings was 0.894, and the IoU for buildings was 0.880. The best model was the one in the final epoch; the learning rate was 0.001176 and the batch norm momentum was 0.01.

We compare the results of our model that of the D-FCN model [8], which is a multi-class algorithm trained on the data fusion 2019 dataset [9], which is a dataset of aerial LiDAR data. We also compare our results to a PointNet++ model also trained on the data fusion 2019 dataset. The results of our model are comparable to but worse than the results of the D-FCN model, which achieves an overall accuracy of 95.6%. However, our model outperforms the PointNet++ multiclass model mentioned in the D-FCN paper, which achieves an overall accuracy of 92.7%. This shows the effectiveness of a binary segmentation model in the context of improving building reconstruction quality.

Additionally, further hyperparameter tuning could be performed to improve the results of the model through a grid search or similar method. With those improvements in mind a better comparison to the D-FCN model could be made. We leave these opportunities for future work and rather present this as a proof-of-concept for the effectiveness of a binary segmentation model in the context of improving building reconstruction quality.

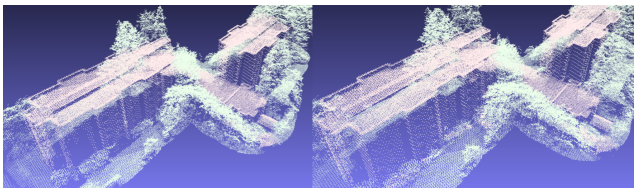

Figure 4. Top: Ground truth segmentation of buildings (red) and non-buildings (green). Bottom: Semantic Segmentation of buildings (red) and non-buildings (green) by the model. The model fills in gaps in the building walls where data is available.

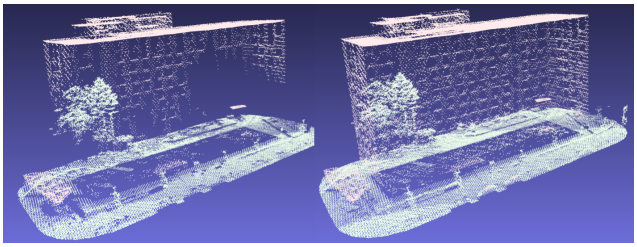

Figure 5. Ground truth (left) of labelled data, and merged segmentation (right) of the labelled data and unlabelled data.

Qualitative Results. The results of the segmentation task are shown in Figure 4. Overall, the model does an excellent job of semantically segmenting medium-sized buildings and smaller. Most of the incorrectly classified points are speckle noise; noise that would get ignored in the voxelization process given that the voxelization considers the most common label in the block [1].

When performing the model on only the unclassified data, the model is still able to segment the buildings with a high degree of visual accuracy. With the entire point cloud considered, the model performs even better and the additional context leads to a better result. Additionally, the model effectively solves the problem of missing walls in the voxelization process, as seen in Figure 5. This greatly improves the reconstruction quality of the buildings, as the voxelization process is able to capture the entire building and not just the parts that are labelled. While further improvements to the model through hyperparameter tuning could still be made to improve the segmentation accuracy, the model is already able to capture the vast majority of the buildings in the point cloud and improve the reconstruction quality of the buildings.

Limitations. The model is occasionally unable to segment very large buildings, as seen in Figure 6. This is due to the lack of context in the block, and the model being un-

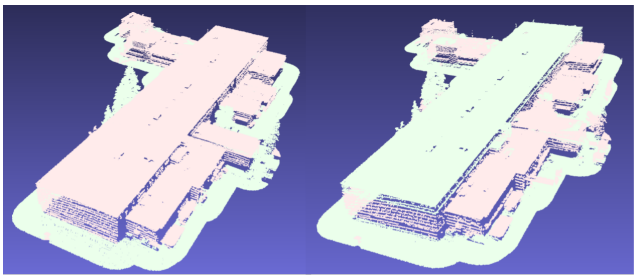

Figure 6. Semantic segmentation of a very large building, where the model incorrectly labels the roof as non-building.

able to extrapolate the local features of the building. Due to the significant variation in the size of buildings in the dataset, simply choosing a larger block size may not be sufficient to solve this issue. With the same number of points, a larger block size leads to increased sparsity in the input data. But with a large number of points, smaller buildings may not have enough points overall to be classified effectively. Such a trade-off is difficult to resolve, and so the model is unable to segment the roofs of some very large buildings. This is not a significant limitation, as it is typically in the walls where large chunks of the building are unlabelled in the original point cloud. As seen in Figure 6, the walls are still classified effectively. This issue can be somewhat mitigated in the downstream task by taking the union of the learned building points and the points already classified as buildings.

4.2. Densification

Qualitative Results. The densification algorithm is performed on the union of the learned building points and the points already classified as buildings. Overall, the densification algorithm works well enough to upsample walls and roofs, as seen in Figure 7.

Limitations. Inherent limitations in the SAPCU algorithm lead to the densification algorithm being unable to create a high quality point cloud for downstream tasks. Given the algorithm was trained on watertight meshes, it does fully preserve the structure of holes in the mesh. Windows are thus not preserved effectively in the densification process. In Figure 7, the densification algorithm is unable to preserve the windows in the building, and many of the windows are either smaller or no longer have their edges preserved effectively. Additionally, the noise caused by incorrect semantic segmentation is significantly amplified after upsampling. This leads to large artefacts in most of the densified buildings, as seen in Figure 7.

An additional drawback with the recursive algorithm is that it leads to differences in the density of the point cloud based on the median split. The number of sampled points was chosen to be 2x the number of input points throughout

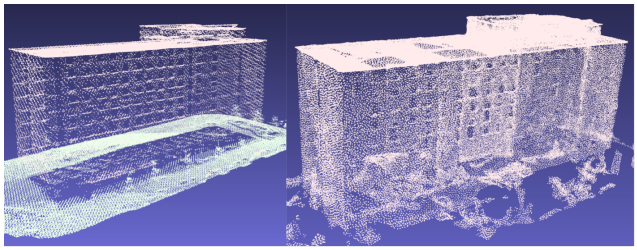

Figure 7. Original labelled point cloud (left) and densification of the labelled point cloud (right). Though upsampling walls and roofs, the model amplifies noise from the segmentation task in the labelled point cloud.

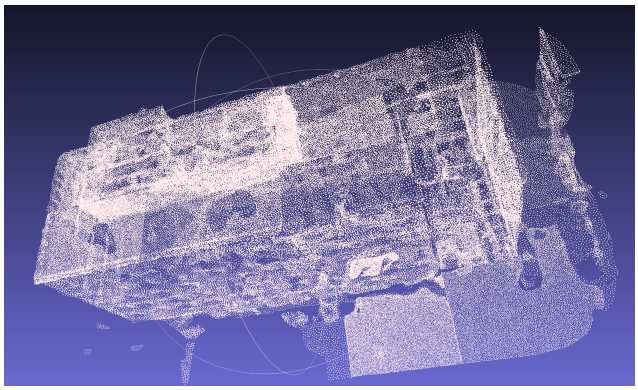

Figure 8. Visible split along recursive median lines in the densified point cloud. Split lines are visible both in ground data and building data

the entire algorithm, however differences in overall density through this median split will be amplified. This is shown in Figure 8, where the densification algorithm is unable to upsample the building to the same extent throughout the building.

4.3. Planar Flattening

Due to time constraints, the planar flattening task is not implemented through code, so no results are yet available. Planar flattening is expected to reduce the noise in the voxelization process, and improve voxelization quality. It would also be useful to perform the planar flattening task on the non-densified point cloud to assess its effectiveness in improving the densification process.

A full ablation study was not performed as a result of the planar flattening process being left for future work. However, the planar flattening process is expected to improve downstream tasks, as it reduces noise in the point cloud and improves voxelization quality.

5. Conclusion

LiDAR-DenseSeg represents an advancement in processing airborne LiDAR data for voxelization and mesh reconstruction. The framework's ability to segment, densify, and reduce variance in the point cloud results in a more accurate and detailed reconstruction of physical structures, particularly buildings. While the implementation showcases notable improvements in handling sparsity and noise in LiDAR data, certain limitations such as challenges in semantically segmenting very large buildings and densifying non-watertight meshes were identified. Future work can explore refining these aspects, potentially enhancing the framework's robustness and applicability, as well as continuation of the planar flattening task. Overall, LiDAR-DenseSeg stands as a promising solution for the accurate and efficient reconstruction of complex environments from LiDAR data.

6. Acknowledgements

The dataloader for PointNet++ is somewhat based on the S3DIS dataloader in the PyTorch implementation of PointNet++ [2, 10].

The recursive median SAPCU algorithm is an extension on the original implementation of the SAPCU architecture [3].

References

[1] A. Mistal, "Minecraftubc: A proof-of-concept for transforming ubc's vancouver campus into a minecraft world through the use of geospatial data processing and machine learning," https://github.com/ashtanmistal/minecraftUBC/releases/tag/1.0.0, 2023. 1, 2, 3, 5

[2] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, "Pointnet++: Deep hierarchical feature learning on point sets in a metric space," 2017. 1, 2, 3, 7

[3] W. Zhao, X. Liu, Z. Zhong, J. Jian, W. Gao, G. Li, and X. Ji, "Self-supervised arbitrary-scale point clouds upsampling via implicit neural representation," in *Proceedings IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, June 2022, pp. 1999–2007. 1, 2, 4, 7

[4] A. X. Chang, T. Funkhouser, L. Guibas, P. Hanrahan, Q. Huang, Z. Li, S. Savarese, M. Savva, S. Song, H. Su, J. Xiao, L. Yi, and F. Yu, "Shapenet: An information-rich 3d model repository," 2015. 2

[5] L. Mescheder, M. Oechsle, M. Niemeyer, S. Nowozin, and A. Geiger, "Occupancy networks: Learning 3d reconstruction in function space," in *Proceedings IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2019. 2

[6] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon, "Dynamic graph cnn for learning on point clouds," *ACM Transactions on Graphics (TOG)*, 2019. 2

[7] U. Library, "Ubc open geospatial data," https://github.com/UBCGeodata/ubc-geospatial-opendata, 2023. 3

[8] C. Wen, L. Yang, X. Li, L. Peng, and T. Chi, "Directionally constrained fully convolutional neural network for airborne lidar point cloud classification," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 162, pp. 50–62, 2020. 5

[9] B. Le Saux, N. Yokoya, R. Hänsch, and M. Brown, "Data fusion contest 2019 (dfc2019)," 2019. [Online]. Available: https://dx.doi.org/10.21227/c6tm-vw12 5

[10] X. Yan, "Pointnet/pointnet++ pytorch," https://github.com/yanx27/Pointnet_Pointnet2_pytorch, 2019. 7