A Appendix

A.1 Datasheet

Table 4 shows the datasheet of RealCity3D. The current RealCity3D contains more than 1,000,000 geo-referenced objects of New York City and Zurich. The four different representations are provided for each object, including polygon meshes, triangle meshes, point clouds, and voxel grids. These data are avaliable at: https://github.com/ai4ce/RealCity3D

Table 4: RealCity3D Datasheet

City	Country	Continent	Raw Meshes	Voxel	Triangulated Meshes	Point Cloud
New York City	United States	North America	download	download	download	download
Zurich	Switzerland	Europe	download	download	download	download

A.2 Author statement

The authors of this paper bear all responsibility in case of violation of rights, including the necessary resolution of such violations. The dataset contains no personally identifiable information or offensive content. The RealCity3D dataset is permitted for use under the Creative Commons Attribution 4.0 International license. The existing CityGML datasets which RealCity3D is based on are licensed under: 1) New York City - NYC Open Data Policies, and 2) Zurich - CC0 1.0 Universal (CC0 1.0) Public Domain Dedication.

A.3 Hosting, licensing, and maintenance plan

The authors are committed to ensuring the RealCity3D dataset remain accessible to public through the RealCity3D GitHub repository and strive to provide the necessary maintenance including manual edits and batch updates. Dataset maintenance will be rigorously logged in human-readable and searchable log files. Public use of the RealCity3D dataset is managed under the Creative Commons Attribution 4.0 International license.

A.4 Auto-encoding Tree

The AETree model is a custom tree-shaped auto-encoder network that uses hierarchically-constructed areal spatial data (Section A.4.1) to encode, decode, and generate city layouts and buildings (Section A.4.2). Figure 10 gives an overview of the method by taking 2D parcels data as an example.

A.4.1 Discovering Spatial Hierarchy in Data

Let us first consider a set of spatial data $\{\mathcal{P}_i\}\ |_{i=1,\dots,N}$, where $\mathcal{P}_i$ represents a single object instance in the set and N is the number of objects. As explained, we focus on 3D cuboids such as buildings, so $\mathcal{P} = (x, y, l, w, h, a) \in \mathbb{R}^6$, where x and y denote the center coordinates of a cuboid, and l, w, h and a denote the length, width, height and orientation angle of the cuboid.

To organize the data hierarchically in a binary tree $\mathcal{T}$, we apply hierarchical clustering [14] by introducing $N-1$ intermediate nodes so that all the original objects stay on the leaf nodes. Concretely, the binary tree is built by recursively merging two closest nodes into a parent node until we obtain a single root node.

The tree is homogeneous, so the intermediate nodes also represent 3D cuboids. For any intermediate node produced, its parameters x and y are obtained as the mean value of corresponding children nodes, and l, w, h and a are defined as the minimum bounding rectangle of its children nodes. Note that before feeding the data into our model, we choose to represent all node parameters *relative to*

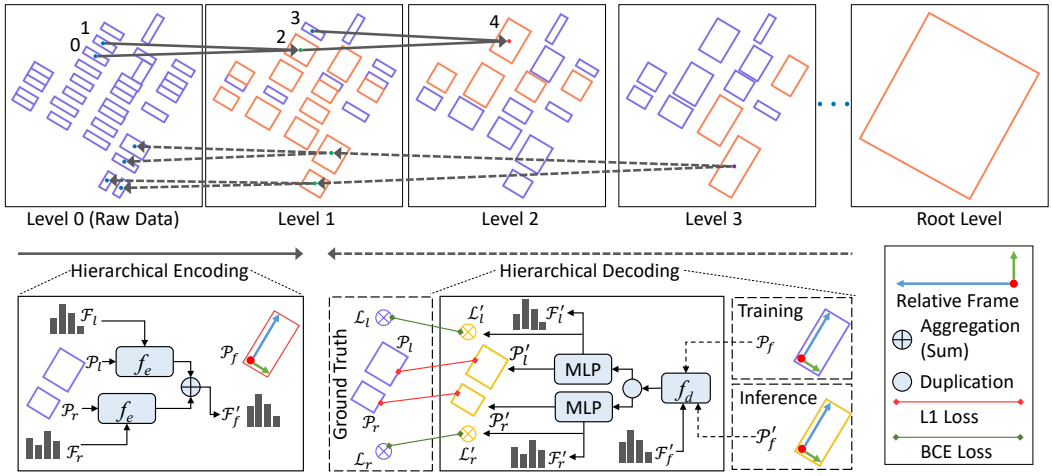

Figure 10: Illustration of the AETree model on 2D city parcels data. The top row in the figure displays an example of pre-calculated tree structure from raw data level to root level. The orange boxes at the each level represent new boxes obtained by merging children boxes form the last level (for example, box 2 in level 1 is generated based on box 0 and 1 from level 0). The bottom row presents our encoding and decoding modules. Based on the tree structure, we first hierarchically encode children nodes to acquire the features of their parent nodes until getting the root node features. Then starting from the root level, we hierarchically decode parent nodes to children nodes and finally obtain the parameter of raw nodes.

their parent nodes (except the root node) as follows (subscript c/f means child/parent),

$$
\begin{aligned}
x_c^r &= \frac{x_c - x_f}{l_f}, \quad y_c^r = \frac{y_c - y_f}{w_f}, \\
l_c^r &= \frac{l_c}{l_f}, \quad w_c^r = \frac{w_c}{w_f}, \\
h_c^r &= \frac{h_c}{h_f}, \quad a_c^r = a_c - a_f.
\end{aligned}
$$

We find this relative representation performs better in reconstruction, and it is only possible in this tree-based (instead of set/sequence) structure (more analysis in supplementary).

The distance metric is also important for hierarchical clustering. The one we use is defined as:

$$
\begin{aligned}
D(i,j) =& \lambda_1 D_{\text{center}}(i,j) + \lambda_2 D_{\text{area}}(i,j) + \lambda_3 D_{\text{shape}}(i,j) \\
&+ \lambda_4 D_{\text{angle}}(i,j) + \lambda_5 D_{\text{merge}}(i,j),
\end{aligned} \tag{2}
$$

where $D(i,j)$ represents the distance between cuboid i and j, λ represents the weight of each distance. Specifically, D_{center} measures the Euclidean distance between the center points of two cuboids; D_{area}, D_{shape} and D_{angle} separately measure the difference between the area, the aspect ratio, and the orientation of two cuboids; and D_{merge} measures the difference between the sum of the two cuboids area and their minimum bounding rectangle area.

We present a simple example to explain our data structure. As shown in Figure 11, three trees are all produced based on four leaf nodes, but through different merging patterns. We introduce an index matrix I_l to store the merging rules at each level l, where in each row, the first two columns denote the indices of children nodes and the last column is the index of the parent node. In this way, we can gather node features of all trees at the same level effectively, and put them in a mini-batch for training and inference.

A.4.2 Tree-shaped Auto-encoder Network

With the data constructed hierarchically, we can naturally develop an auto-encoding neural network with a tree shape to encode, decode and generate the cuboid sets. Our encoder learns a latent

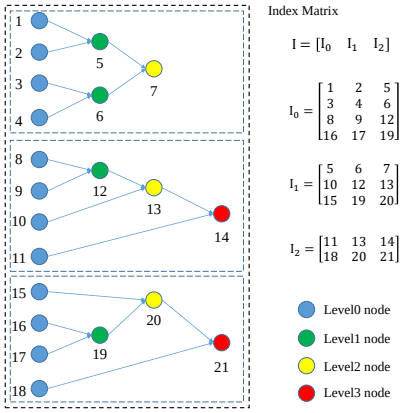

Figure 11: An example of our data structure. The blue nodes at level 0 store raw objects and we build a hierarchical clustering tree according to their pairwise distances. For efficient mini-batch training, we design an index matrix for each level storing the indexes of children and parents.

representation $\mathcal{F}_{root}$ of the root node by encoding each node from bottom to top. Conversely, our decoder decodes the root node from top to bottom and reconstructs the original data. Each node in the tree is represented by its geometric parameters $\mathcal{P}$ and a feature representation $\mathcal{F}$.

Encoding. To obtain the root node feature $\mathcal{F}_{root}$, the encoder encodes all intermediate nodes from bottom to top level by level. Given the the parameters and features of a left child and a right child, the feature of a parent node is computed as:

$$\mathcal{F}'_f = f_e([\mathcal{P}_l, \mathcal{F}_l]) + f_e([\mathcal{P}_r, \mathcal{F}_r]), \tag{3}$$

where f_e represents an encoding function that encodes children parameters and features. In the experiments, we tried both MLP and LSTM cell as the f_e function. And it can be seen that our encoding function is symmetric, meaning the encoded parent feature does not contain order information. Note that the parameters of parent nodes $\mathcal{P}_f$ are pre-computed during data construction, and the feature values of the leaf nodes are initialized as zeros.

Decoding. The decoder aims to reconstruct the original data from the root features $\mathcal{F}_{root}$ produced by the encoder. At each level, we decode from a parent node into two children nodes, which can be formulated as

$$[\mathcal{P}'_l, \mathcal{F}'_l, \mathcal{L}'_l, \mathcal{P}'_r, \mathcal{F}'_r, \mathcal{L}'_r] = f_d([\mathcal{P}'_f, \mathcal{F}'_f]) \tag{4}$$

where f_d denotes the decoding function, $\mathcal{P}'$, $\mathcal{F}'$ and $\mathcal{L}'$ indicate a node's decoded parameters, features, and indicator of being leaf nodes or not. We add this indicator judgement to determine whether the current node should be further decoded or not at inference time. Whenever all the nodes are identified as leaf nodes, the decoding process will stop.

During training, following the idea of teacher forcing, we use the ground-truth (pre-calculated) parameters $\mathcal{P}_f$ of the parent node as the decoder input, which renders the model training faster and more efficient. L1 loss and Binary Cross-Entropy (BCE) loss are used to minimize the errors of predicted parameters and indicators, respectively.

As introduced in above paragraphs, we employ both MLP and LSTM cell as the f_e function to hierarchically encode features from bottom to top. The details of two different encoding modules can be found in Figure 12. For the encoding module of AETree(MLP) (Figure 12 left), we adopted a shared MLP to learn features of the left and right nodes. And then the features of the father nodes ($\mathcal{F}'_f$) are obtained by using a summation aggregation function on learned features. Moreover, for encoding module of AETree(LSTM) (Figure 12 right), we first split the features of children nodes to hidden states(h) and cell states(c), which are input into a LSTM Cell along with the parameters of nodes($\mathcal{P}$). The output of the LSTM Cell are concatenated at each node and then summarized to obtain the features of the father nodes($\mathcal{F}'_f$).

Similarly, we illustrate the decoding module of AETree(MLP) and AETree(LSTM) in Figure 13. For AETree(MLP) model, the learned features and parameters of nodes are input to its decoding

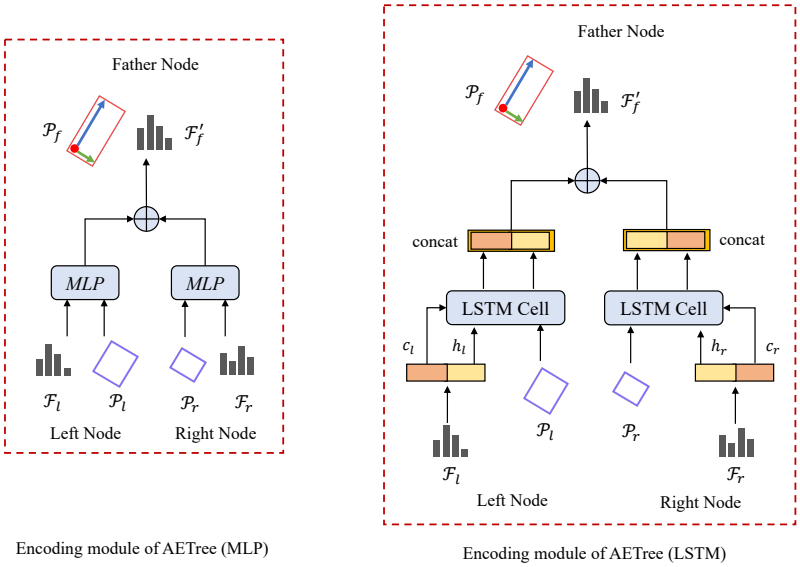

Figure 12: Illustration of the encoding module of AETree(MLP) and AETree(LSTM).

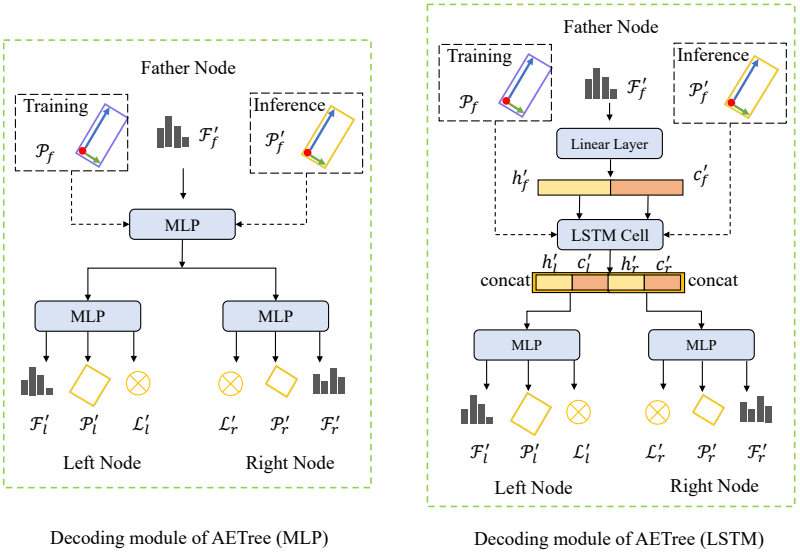

Figure 13: Illustration of the decoding module of AETree(MLP) and AETree(LSTM).

module. The inputs first pass through a MLP layer to enlarge the feature dimensions and then go through another MLP layer to acquire the parameters, features, and indicators of children nodes. And for AETree(LSTM)'s decoding module, we also implement a linear layer to enlarge the feature dimensions and then split features to hidden states(h) and cell states(c). By inputting hidden states, cell states and the parameters to a LSTM Cell, we divide the output of hidden states and cell states into two parts, where one part of hidden states(h'_l) and cell states(c'_l) are concatenated as the intermediate features of the left nodes, and the other part of these two states are concatenated as the intermediate features of the right node. The parameters, features, and the indicator of each node are finally acquired by employing the MLP on the corresponding intermediate features.

Generation. To empower the model with data generation capability, we fit a Gaussian Mixture Model (GMM) on the root feature representation. Specifically, we obtain the root features of all the

training data by passing into our encoder, and then estimate this distribution with a GMM. During the data generation process, we sample a new root feature $\mathcal{F}_g$ from the fitted GMM distribution, and a new data is generated going through our decoder.

Latent Space Interpolation. Given latent representations of two box sets, we can obtain the intermediate box set by applying the decoder to the linear interpolation between these two latent spaces. Figure 14 shows the reconstructed box sets from the interpolated latent vectors. Interestingly, we produce a gradually varied sequence of box set from box set S to box set T, which demonstrates the smoothness of our latent space. More importantly, this indicates that our learned latent representations are generative instead of simply memorizing the training sets.

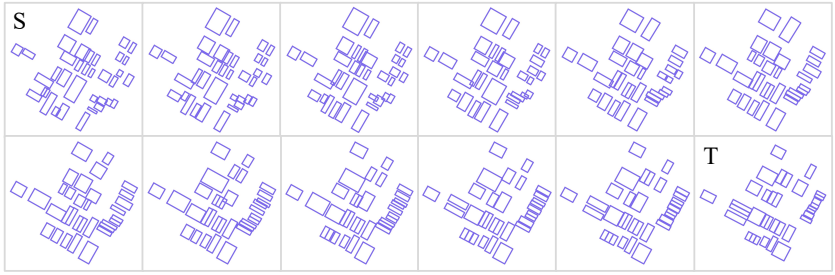

Figure 14: Latent space interpolation between two box sets, S and T, using AETree.

A.5 Results of Building Shape Generation

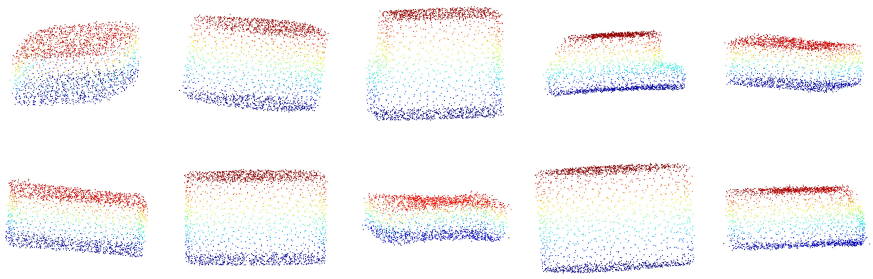

Figure 15: Some shape generation results of Latent-GAN [3] trained on RealCity3D. It can be seen that the reconstructions lost many important geometric details and variations of the 3D building shapes. This suggests again that the RealCity3D dataset is non-trivial.

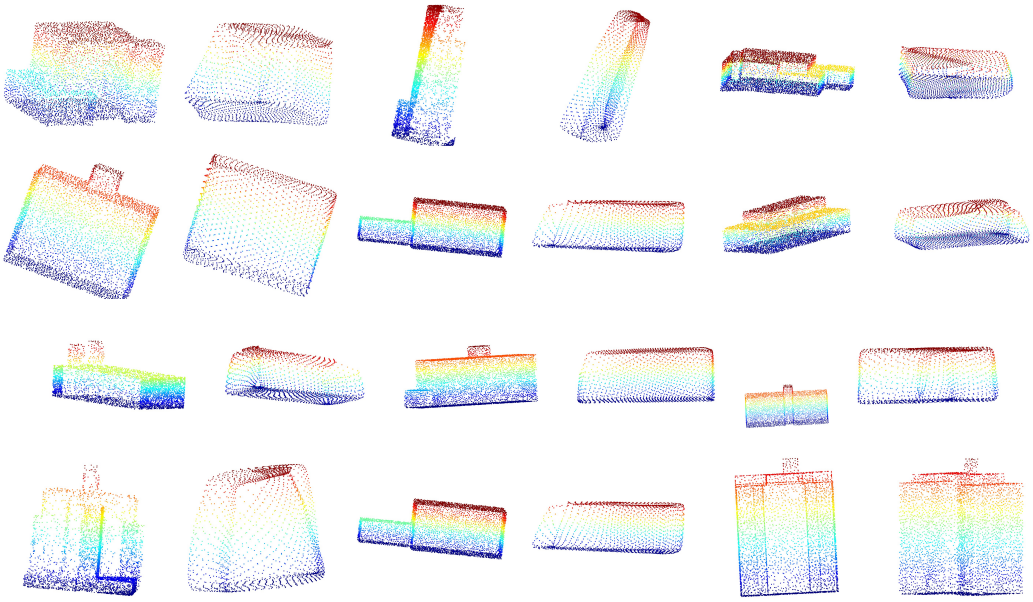

Figure 16: Some testing results of FoldingNet [30] trained on RealCity3D. For each pair of shapes, the first one is the input point cloud and the second one is the auto-encoder's reconstructed point cloud. It can be seen that the reconstructions lost many important geometric details of the 3D building shapes. This suggests that the RealCity3D is a non-trivial 3D shape dataset.