Generating Realistic and Diverse Tests for LiDAR Perception Systems Full Implementation Details

Each mutation is implemented using Numpy [2], SemanticKITTI's developer tools [1], and open3D [3], a popular open source point cloud manipulation library for Python.

A. Building the Resource Set

When building the resource set, the filter to identify entities that are too small, too far from the sensor, or occluded are parameterized as follows. First, the entity must contain more than $minPointThreshold$ points and the distance to the entity must be less than $distThresh$. Second, to filter occluded objects, the bounding box of the entity is acquired and scaled based on the distance from the sensor by a factor of at most $occlBuffer$ to provide a buffer to ensure all occluding objects are identified. The box is also extended to the ground in case another entity is occluding the bottom portion of the object, for example, a car that is only comprised of the roof points. A frustum (bounding cone) is drawn from the sensor to the altered bounding box and if any points that are not semantically classified as unlabeled, outlier, or some kind of ground (road, sidewalk, etc) are included within this new polygon the entity is discarded.

1) Entity Identification: As described in Section -A, the tool performs Entity Identification to automatically identify entities not captured in the data set. Because SemanticKITTI does not capture sign entities, the implementation only seeks to identify sign entities. To accomplish this, the points labelled as poles and signs are isolated and grouped into clusters using open3D's DBSCAN (Density-Based Spatial Clustering of Applications with Noise). If a cluster has both sign points and pole points, has more than $minSignPoints$ points, is smaller than $signSizeThreshold$, and meets the other resource set criteria, it is classified as a unique sign entity and added to the entity list.

B. Add Object

An entity is added into a scene at a given location by adding its points at that location and then removing its shadow. To ensure that the entity is viewed from the correct perspective, the distance from the LiDAR sensor is kept consistent. When adding an entity, $RealInv$ verifies that the entity does not intersect another entity by voxelizing the non-ground points and comparing this to the entity's new location. Then, the entity is checked for occlusion by drawing a frustum from the entity's bounding box to the sensor and ensuring that there are fewer than $addOcclusionThreshold$ points in the frustum. Finally, $RealInv$ verifies that the entity is being placed on the ground in a location that is semantically viable. To evaluate this, the viable ground points are flattened and voxelized and a portion of the entity's points greater than the $groundThreshold$ must be above the ground voxels. Due to the mounting position and angle of the LiDAR, the ground may not be visible within a short distance of the sensor. To accommodate, if the entity is closer than the $groundDistanceThreshold$, this check is ignored.

C. Remove Entity

To remove the selected entity from the scene, the shadow removal process is run in reverse. First, the the entity's shadow is generated—this shows the area that must be filled to replace the entity. The shadow area is then split in two halves vertically, and each half is swept rotationally outward from the entity to identify an equal sized portion adjacent to the entity that could be used to back-fill the vacant area. These points are then copied into the area, preserving the distance of the points and semantics surrounding the missing area. During this process, $RealInv$ verifies that the entity to be removed does not partially obscure another entity and that the points that will fill in the empty area have appropriate semantics. To avoid obscuring another entity, there must be fewer than $removePointsAboveThresh$ points above the entity. To ensure appropriate semantics of the filled area, the filled points must not contain points labeled in the $invalidReplacementClasses$ set. These classes are chosen since they are likely to result in a non-conforming replacement; for example, if there are building points in the replacement, the building will be disconnected from its original instance.

D. Rotate Entity about Sensor

Used in conjunction with the Add Entity mutation, this mutation is used to identify a location within a scene where an entity could be placed and align the entity with that location, while $RealInv$ ensures that it is not occluded by scenery. To do so, a line is drawn from the sensor to the entity's center at the rotated position. The non-road points are voxelized and if the line intersects with any of these voxels, the mutation is not performed. If valid, the entity is aligned with the road at that position.

E. Mirror Object

Mirroring an object is accomplished by flipping its points about the vertical axis. Mirroring an object has no specific invariants, though the newly mirrored entity may be subject to additional invariants in later mutation stages.

F. Adjust Intensity Mutation

This mutation directly adjusts the intensity of the selected entity's points in place. First, $RealInv$ ensures that surfaces that cannot be altered are left untouched by clustering the points of the entity based on intensity into two groups and removes any points that exceed a threshold greater than $intensityMutateThresh$ and the larger grouping is selected. This is done as high intensity areas correspond to surfaces that are unlikely to be altered in the physical world, e.g. there are laws against obscuring license plates, headlights, etc. The parameter for the amount to adjust the intensity is randomly selected between $minIntensityThresh$ and $maxIntensityThresh$. If the average intensity is greater than $intensitySubThresh$ then the intensity is decreased by the selected amount, otherwise it is increased by the selected amount Although this mutation is generally applicable to multiple entity classes, we specifically explore adjusting the intensity of vehicles in our implementation for the physical parallel of re-painting a vehicle.

G. Deform Mutation

This mutation directly adjusts the selected entity's points in place to induce a deformation. One of the entity's points is randomly chosen, then a the number of points to deform, K, is chosen between $minDeformThresh$ and $maxDeformThresh\%$ in accordance with $RealInv$. The closest K points to the main point are found using open3D's kd-tree, then a deformation amount for each point is randomly generated with a mean of $deformNoiseMean$ and standard deviation of $deformNoiseStdDev$ in accordance with $RealInv$. The generated noise is sorted in descending order and then each point is pushed that amount away from the sensor, assigning by how close each point is to the main point, forming a convex hull at the selected location. Although this mutation is applicable to many entity classes as described in the approach, we specifically explore applying deformations to vehicles in our implementation.

H. Scale Mutation

This mutation directly adjusts the selected entity's points in place to induce a change in scale. When adjusting an entity's scale, $RealInv$ verifies that the entity has more than $minScalePointThreshold$ points in order to ensure the entity contains enough definition to be identified by the perception system. Additionally, although not related to realism, an invariant is imposed that the entity have fewer than $maxScalePointsThreshold$ due to the large computational cost for larger entities. Further, to ensure that the mutation is impactful, the increased scale should occlude points not occluded by the unscaled entity.

To alter an entity's scale while preserve the shape, the tool first creates a hull for the entity using open3D's ball pivoting algorithm in order to create a mesh that captures the holes that can occur on entities, such as the front windshield of a vehicle. The resulting mesh is smoothed and then a scale of $scaleAmount\%$ in accordance with $RealInv$ is applied to the mesh. Any of the original points that intersect the new mesh are brought to the intersection point on the mesh. The original points are scaled and then a nearest-neighbor match is applied between the new points and original points to provide the new points with an intensity that is as close as possible to the original entity's intensity. Although this mutation is general and can apply to multiple entity classes, we specifically explore the scaling of vehicles in our implementation.

I. Class-Preserving Swap: Signs

Although the approach to this mutation is general, we specifically explore swapping one type of sign for another for our implementation. To accomplish this, we create five sign meshes based on the dimensions of their real life counter parts: crossbuck, speed limit, stop, warning, and yield. The mesh of the replacement is aligned to the sign pole's center, then rotated to match the angle of the original sign. This is done by matching the mesh's left and right points to the original sign's left most and right most points from the perspective of the LiDAR sensor. Then all points that intersect with the sign mesh are brought to the mesh. Nearest neighbors matching is used between the new sign points and the original sign points to obtain the closest intensity. During this process, $RealInv$ verifies that the sign choosen to be replaced has at least $minSignPointThreshold$ points to ensure it has enough definition to be mutated. Further, $RealInv$ verifies that the swapped sign appears at an appropriate height of at least $minSignHeight$ and is similar in size to the original sign, being within $signSizeAllowance$ across the longest dimension. Similar to the scale mutation, to ensure that the mutation is impactful, the swapped sign should occlude points not occluded by the original sign.

REFERENCES

[1] J. Behley, M. Garbade, A. Milioto, J. Quenzel, S. Behnke, C. Stachniss, and J. Gall. SemanticKITTI: A Dataset for Semantic Scene Understanding of LiDAR Sequences. In *Proc. of the IEEE/CVF International Conf. on Computer Vision (ICCV)*, 2019.

[2] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[3] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A modern library for 3D data processing. *arXiv:1801.09847*, 2018.