Extending Scenic to Generate Three-Dimensional Meshes of Road Networks

ETHAN MAKISHIMA

ACM Reference Format:

Ethan Makishima. 2025. Extending Scenic to Generate Three-Dimensional Meshes of Road Networks . 1, 1 (August 2025), 7 pages.

1 INTRODUCTION

Scenic[3] is a probabilistic programming language designed to create and evaluate scenarios for perception systems in autonomous vehicles and other cyber-physical platforms, such as robotics. It generates "scenes" composed of objects and agents, allowing users to repeatedly simulate diverse environments. By producing numerous variations, including rare or edge cases, Scenic supports machine learning workflows, helps identify potential failure modes, and enables rigorous testing and validation of perception-driven systems.

Currently, Scenic supports only two-dimensional polygonal map representations, preventing accurate modeling of three-dimensional road features such as ramps, bridges, hilly roads, and tunnels. To address this limitation, we will extend Scenic's codebase to parse three-dimensional elements and generate corresponding 3D meshes.

Scenic's two-dimensional implementation parses OpenDrive (XODR) [1] files to extract all road elements. These OpenDrive files contain an extensive list of data about the road network, including the different types of roads, position, and shape. Scenic's parser reads each road's reference points along with their left and right boundary coordinates. For every segment, the parser generates transverse lines at each centerline point and extends them outward to the specified boundaries. Once all individual section polygons are created, they are unioned, or combined together, into a single, continuous road polygon. The parser would then take these polygons and convert them into Scenic objects. We use the class called 'NetworkElement', which is a class that aggregates the shape and dimensions together. Once all the network elements have been created, we combine the polygons together into one Scenic Network, which aggregates all the combined meshes.

To implement the parsing of three dimensions into Scenic, we will do the following:

- Modify the calculations in the parser file
- Build meshes instead of polygons
- Modify the Road objects to include new geometries and different classes of regions.
- Create three-dimensional visualizations of the road map scenarios

Author's address: Ethan Makishima.

Manuscript submitted to ACM

- Create several simple tests to verify our implementation

2 SOLUTION APPROACH

2.1 Parsing File

The parsing file takes an OpenDrive (XODR) file and gets all the elements, such as roads, lanes, groups, dimensions, and what we are implementing, the elevation. Firstly, we need to parse all elevation elements from each elevationProfile element. Each elevation element contains doubles (s, a, b, c, d) such that s is the s-coordinate of the start position, while the rest of the variables correspond to the following equation: $elev(ds) = a + b * ds + c * ds^2 + d * ds^3$. In the equation, ds is the distance along the road reference line between the start of a new elevation element and the given position. The variable ds starts at zero for each element. (Fig. 1)

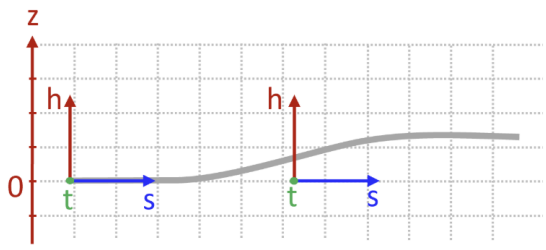

Fig. 1. Elevation along the s-coordinates

2.2 Calculating the new reference points and geometry:

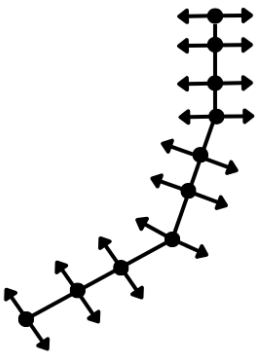

Fig. 2. The centerline of the road, with reference points along the line with tangent line to expand outward to form the left and right bounds.

Now that we have our elevation element for each road, we can calculate the z-coordinate for each reference point. The reference points for a road are sampled from the centerline and built from from the reference points outward toward the left and right bounds (Fig. 2). To implement the elevation, we can take the existing function to calculate the

reference points and plug in our elevation equation to get the z-coordinate at each reference point.

Now that we have the z-coordinates, we need to build meshes in place of the polygons for roads. For this, we will use the Trimesh [4] library to build these objects. For every right and left bounds for each reference point i, we will build a Trimesh mesh with two faces, one with left i to left i + 1 to right i, and one with right i to left i + 1 to right i + 1 (Fig. 3). Then, we combine our meshes together to get our complete road mesh by concatenating the meshes.

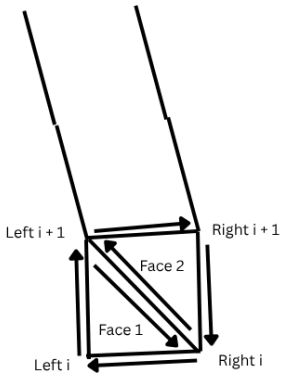

Fig. 3. Shows the construction of the two faces of a road segment.

2.3 Converting Roads into Scenic Network Elements:

Once we have all the meshes, we can move on to generating the Scenic network elements. We define the Scenic elements with the class 'NetworkElement', which is an aggregation of the geometries, dimensions, and other attributes that define the object. Scenic elements such as the road, lane, shoulder, and sidewalk are all built using the 'NetworkElement' class. For building these network elements, we will pass in a mesh instead of a polygon for its shape. The centerline, left edge, and right edge of the network element, which define those geometric features of the element, take in a region object. The Region class is a class of objects that defines a geometric space in Scenic. For the previous implementation, the region object was a polyline region, which is defined as a two-dimensional line. We change this to a path region, which is a three-dimensional line defined by multiple polyline regions. We do a similar modification for the forward and backward lane groups, passing in the concatenated trimesh of either all the forward and backward lane groups for the geometry, and passing in path regions instead of polyline regions for the centerline, left edge, and right edge.

2.4 Converting RoadMap into Scenic Network:

Scenic needs a single mesh for each of the categories of regions (drivable region, sidewalk region, shoulder region, etc.). Once we have created all of the network elements, we can combine them into a single mesh. All existing calculations to build the meshes stay the same, except when we combine the geometries, we use a concatenate function specific to meshes instead of the previous buffer union function, which only works with polygons. So we concatenate the roads

together, the sidewalks together, and the shoulders together into three separate meshes. The main calculation of the intersections for the roadmap stays the same, but similarly, we replace the buffer union function with the concatenate function.

Once all the calculations are done, we build a Scenic network object. Similar to the Scenic network elements, the Scenic network defines the geometries of the entire map through the categories of the network elements. The region object that was used to define each category was a polygonal region, which is an aggregation that defines the dimensions of the group of polygons, is replaced with a mesh surface region, which is another aggregation of the shape, dimensions, and other attributes of the group of meshes.

3 CHALLENGES

In general, I was able to use the existing code for two dimensions to implement three dimensions. Most of the calculations stayed relatively the same, with small changes to variables and types. All variables that dealt with coordinates were simple to change, just changing them from (x, y) to (x, y, z) while making sure the calculations also handle the z-coordinate. All polygon elements were converted to Trimesh meshes, and all buffer unions for the polygons were replaced with Trimesh concatenate functions. As for the Region types, polylines were changed to path regions and polygonal regions were converted to mesh surface regions. Although the bulk of the code stayed relatively the same, some challenges required more modifications.

3.1 NetworkElement class

As previously mentioned, the 'NetworkElement' class defines the object's geometries and dimensions. The 'NetworkElement' class was previously inherited from the polygonal region class, which caused problems when trying to pass in meshes and other region types into the class. To solve this problem, the 'NetworkElement' class now inherits from the general Region class. Also, general region functions that have different implementations across polygonal regions and mesh surface regions are now created within the 'NetworkElement' class and call upon the base region functions. So, for example, the function intersect is now implemented in the 'NetworkElement' class and calls upon Region.intersect(self, other), which will either do the intersect for a polygonal region or mesh surface region, depending on the type of self 4.

```
def intersect(self, other):
    return Region.intersect(self, other)
```

Fig. 4. Code snippet of the intersection function in the NetworkElement class.

3.2 R-tree Implementation

The network uses an R-tree, a tree data structure to store geometrical objects. This is very useful for Scenic as it can query a large number of geometric objects relatively quickly. However, the previous implementation uses a function from the Shapely library [5]. This function, called STRtree, takes in polygons and builds the tree. We can use the same function with meshes by converting them into mesh surface regions and calling the bounding polygon function, which

returns a polygon that bounds the x and y dimensions of the mesh. This will properly build a working R-tree from the meshes and work with the Shapely function STRtree.

4 RESULTS

4.1 Mesh Creation

The meshes are a bit rough, but the general shape and size of the maps are accurate. The simulator CARLA [2] provides several XODR files on which I did most of my testing. I have attached an image of some of the maps that were ran on CARLA's simulator, and provide images of the meshes that I have generated using Scenic. (Fig. 5, Fig. 6)

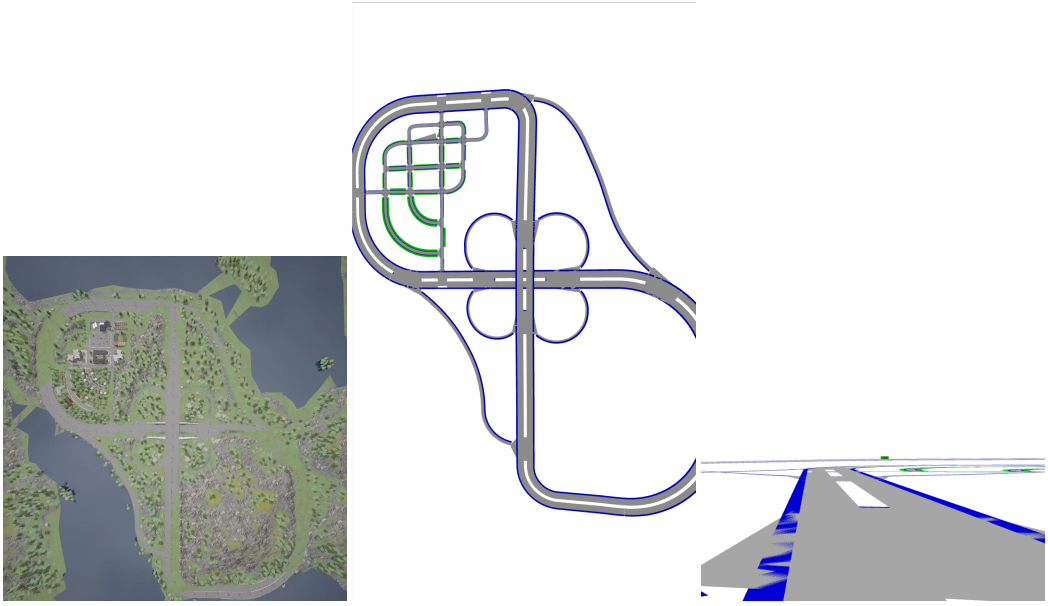

Fig. 5. First Image: CARLA simulation of map Town04, Second Image: Scenic representation of meshes, Third Image: Side view of the overpass

The Scenic-generated meshes are fairly accurate to the CARLA representations, but there are a couple of artifacts that need to be cleaned up. Some of the roads in both maps should have shoulder elements in the center of their road, but just have noticeable holes in the mesh. Also, in Town05, there is a noticeable sidewalk artifact (green area on the left side of the mesh). Future work includes cleaning up the holes and other artifacts in the meshes.

4.2 Generating Scenes

Testing the results also included running some simple Scenic code. We want to test if a car can properly generate on elevated meshes. So, we coded a very simple Scenic script that generates a car in the center of a lane, ensuring that the elevation is greater than 1. We then generate the scenario using the Python API and check the lane that it generates

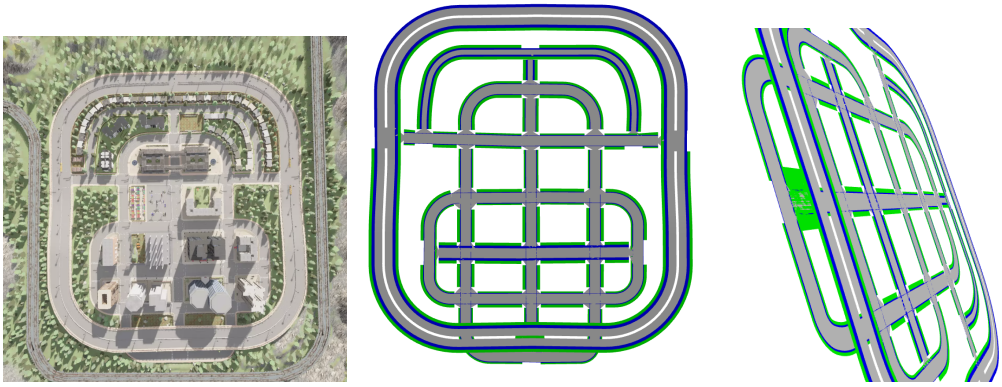

Fig. 6. First Image: CARLA simulation of map Town05, Second Image: Scenic representation of meshes, Third Image: Side view of the map

on. Observing the car's position and the lane it is on, we can see that cars are able to generate on the elevated meshes. Other testing showed that generating a car on any road was able to work properly.

We also tested the generation of a car on the two different roads that shared an x and y coordinate, but had different z-coordinates. Performing this test at Town04, we attempt to generate two cars in the overpass part of the map, one at the bottom of the road and one at the top of the road. We did this by creating two simple Scenic snippets that generated on the same x and y coordinate, but one generates on the z-coordinate of 0, and the other we set the z-coordinate to 100 so that the car will map down to the road. We can generate both cars, but we notice that the top car has the orientation of the bottom car, meaning that the top car is facing perpendicular to the road that it is on. (Fig. 7)

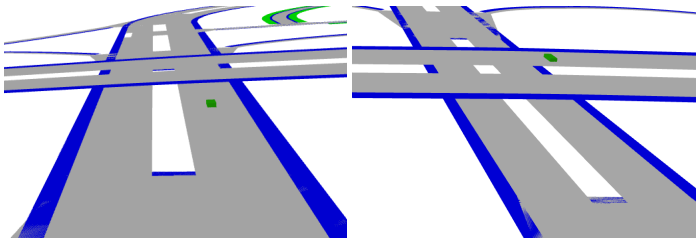

Fig. 7. First Image: Car generating on bottom road of overpass, Second Image: Car generating on top road of overpass

5 CONCLUSIONS AND FUTURE WORK

Scenic is now able to parse OpenDrive files and render a three-dimensional mesh of maps. Although these maps are rough and have room for improvement, they still capture the proper elevations of the roads.

An additional element that still needs to be parsed is the super elevation element, which describes the angle of the slope of the road. Also, the three-dimensional implementation is hard-coded, meaning all the checks to see if you are generating a three-dimensional map or a two-dimensional map are hard-coded to generate three-dimensional maps. So, another thing that can be implemented is to be able to generate meshes and to be able to generate the two-dimensional

polygons if the −2d flag is set.

REFERENCES

[1] ASAM e.V. 2024. *ASAM OpenDRIVE BS 1.8.1 Specification: Static Road Network Description.* Technical Report. ASAM e.V. https://publications.pages.asam.net/standards/ASAM_OpenDRIVE/ASAM_OpenDRIVE_Specification/latest/specification/index.html

[2] CARLA Team. [n. d.]. *CARLA Simulator Documentation.* https://carla.readthedocs.io/en/0.9.15/

[3] Daniel J. Fremont, Edward Kim, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, Sanjit A. Seshia. 2023. Scenic: a language for scenario specification and data generation. *Machine Learning* 112, 01 (Oct. 2023), 1–18. https://doi.org/0.1007/s10994-021-06120-5

[4] Dawson-Haggerty et al. [n. d.]. *Trimesh.* https://trimesh.org/

[5] Shapely Community. [n. d.]. *Shapely: manipulation and analysis of planar geometric objects.* https://shapely.readthedocs.io/