Embedded Systems Design Project
Nvidia: HiL Simulation for Automated Driving

ÖRN ARNAR KARLSSON
SAMUEL TANNER LINDEMER
TONY LUNDGREN
BRATISLAV MARKOVIC
OSKAR NÄSLUND
TOFIK SONONO
PRATHEEK UNNIKRISHNAN

November 26, 2018

Abstract

Contents

List of Figures

List of Tables

Nomenclature

a	First letter in the alphabet
b	Second letter in the alphabet
c	Third letter in the alphabet

1 Introduction

1.1 Background

1.2 Project Description

1.3 Delimitation

1.4 Readers Guide / Report Disposition

1.4.1 Methodology

1.4.2 Implementation

1.4.3 Verification and Validation

1.4.4 Results

1.4.5 Discussions and Conclusion

1.4.6 Future Work

1.4.7 Appendices

2 Literary Review and State of the Art

3 Methodology

3.1 Vector Maps

At the outset of the project, the stakeholder's stated requirement was an automated process for generating an Autoware-compatible vector map for any PreScan simulation. The stakeholder was previously using a labor-intensive approximation method which only generated a very small subset of the vector map.

3.1.1 Requirements Elicitation and Specification

The project group attended a day-long seminar with a TASS International representative to establish alternatives. Four possible approaches were established.

1. **Approach currently used by stakeholder:** Manually set a simulated vehicle to traverse every roadway. Save the location data to a file. (It was stipulated that this process could be automated.)

2. **Approach recommended by TASS International:** Write scripts to automate PreScan's lane detection sensor. Scan the entire road map and save the sensor data to a file.

3. Take screen captures of the road map and use image recognition methods to approximate the roads.

4. **Approach chosen by project group:** Reverse-engineer PreScan's internal road generation methods, as well as its undocumented simulation configuration file.

Options 1 through 3 were identified as the most expedient approaches to the problem, as the developers could capture the road curvatures and dimensions without necessarily understanding anything about PreScan's internal logic. However, these naive approaches came with the risk that future updates to PreScan's MATLAB user interface or sensor definitions would render them inoperable.

It was decided that the feasibility of Option 4 should be investigated before proceeding. PreScan's XML-formatted configuration file is not documented for developers, and TASS International was unable to provide us any other API to directly access the road configurations. The configuration file contains a list of attributes defining road orientations and dimensions, but the methods and equations for generating the roads

are not documented anywhere. These would have to be derived through experimentation. To complicate things further, it was discovered that the attributes listed in the configuration file only define the curvature of the road center lines, but the vector map requires separate curves for the center, lanes and road edges.

Thus, the configuration file approach presented both the highest development risk and the highest reward. The risk in pursuing this approach was the possibility that accurately deriving the complete road definition would not be possible within the time frame. The reward in pursuing this approach was producing a software package package requiring only a single input file which would be resilient to future PreScan updates. (It was judged highly unlikely that PreScan would completely overhaul their configuration file structure in a future software update.) Weighing these risks and benefits, the group concluded that this option was the best approach. After assessing the scale of this sub-project, two group members, Örn and Samuel, decided to make vector maps their entire focus for the reimainder of the project.

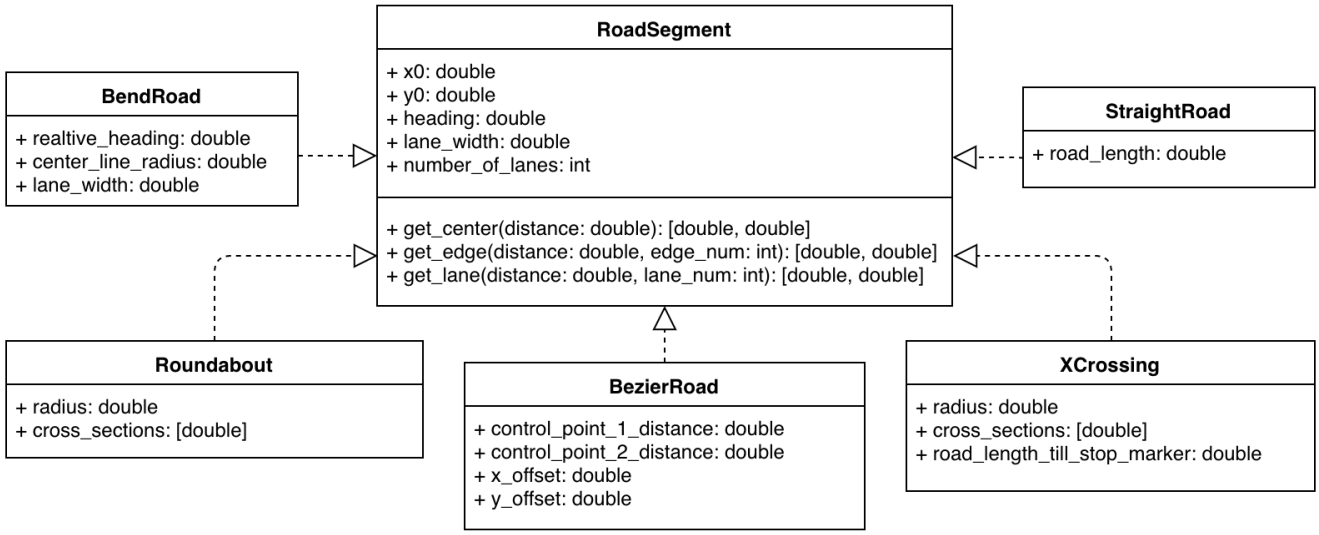


Figure 1: UML class diagram of road data in PreScan's configuration file. The bulk of the information defining the roads are in the algorithms used to generate them from these values. These algorithms are not accessible through the PreScan API.

With the requirements adequately understood, two UML diagrams were drawn: one modeling the PreScan data represenation (see Figure 1) and another modeling the vector map represenation (see Figure 2). The vector map data representation relies on coordinate data and the sequential relationships between them. The system of linking and sequncing every data point requires virtually complete knoweldge of the entire road system before the vector map can be generated. Unfortunately, there is no official documentation available for the vector map system.

3.1.2 Agile and Extreme Programming (XP)

It was clear in the early that the translation process would be fairly complex, and selecting an appropriate software development methodology would be crucial for a good result. It was apparent that our understanding of the system would evolve over time, and designing a complete software model without testing any code first would be difficult. This warranted an Agile software development approach, rather than a more rigid paradigm like the Waterfall or V-model framework. We chose Extreme Programming (XP) for its emphasis on pair programming, interaction and test-driven development. Here we present why we concluded this was the correct approach and how the principles and activities of XP concretely improved our design process.

Pair programming. The early challenges we faced were largely based on mathematical problem solving. Solving these types of problems are often made easier when code can be used to assist dialog between problem solvers. Pair programming initial development steps followed a fairly simple system: one programmer would code a proposed algorithm on generating the roads until all the test input files could be

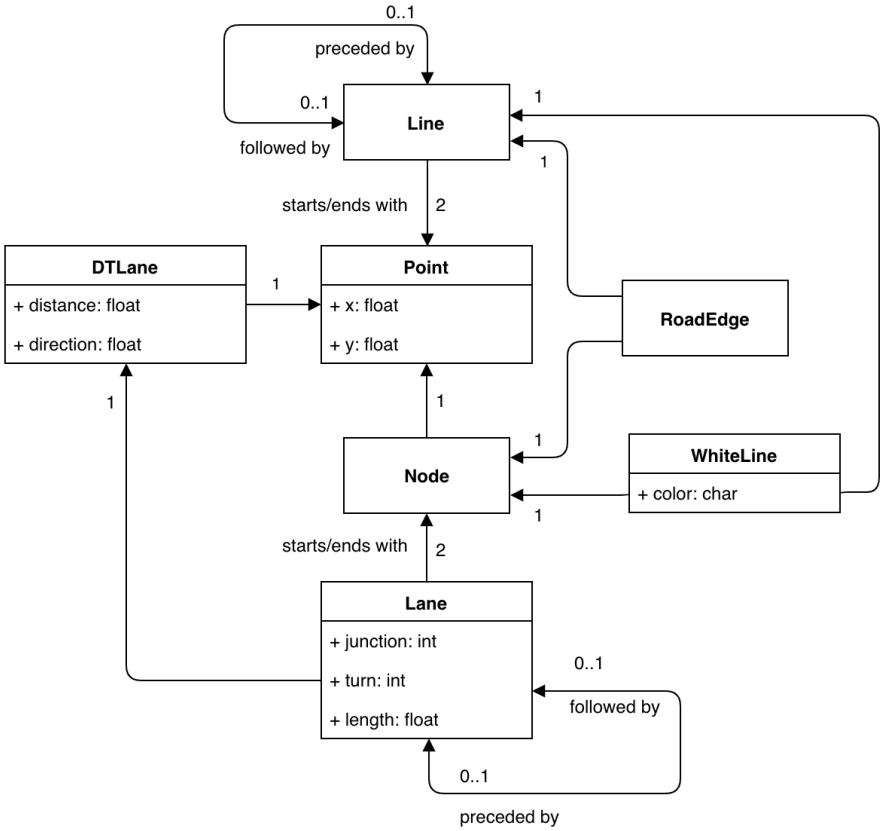


Figure 2: UML class diagram of road data in the vector map format. Drivable lanes and other road features are defined as a sequence of linked coordinates. Each object in the above UML class diagram is represented as a single line in its corresponding CSV file.

parsed without errors. The results would be then be plotted for a visual comparison with the original roads in PreScan. The disparities would be discussed, and the keyboard would change hands.

With this process, many ineffective algorithms were ruled out in a matter of hours without duplicating work. The dialog between the two programmers kept the work flowing for hours at a time where a "mental block" would have been reached had the pair been working independently. Once all of the road models were being generated correctly, the two programmers each had a perfect understanding of how it all worked. The two were then able to quickly come to an agreement on where code could be reused and which interfaces were appropriate for the new classes.

Stand-up meetings. These meetings incorporated members of the project group outside of the two programmers working directly on the vector mapping code. As the programmers justified design decisions on a whiteboard, other group members could offer a fresh perspective on a problem and point out incorrect assumptions or poor design choices. In one such meeting, Samuel explained that the latest version of the vector map generation code could take several minutes to run because every node had to be compared to ensure there were no duplicates. After the design choice that led to this was discussed, Tony demonstrated how clever use of hash tables would make this operation nearly instantaneous.

In another meeting, Örn and Samuel explained how three road models had already been derived without the use of high-level calculus, so no delays related to math should be expected. Oskar then pointed out that the Bezier road model that had been proposed was really only a rough approximation that would work for trivial cases. No test case had been written for non-trivial configurations of Bezier roads up to that point. Upon further research it was discovered that no analytical solution existed to this specific problem, and a numerical algorithm would have to be developed (see Section 4.1.1).

Test-driven development. In XP, large problems are broken into smaller targets defined by a set of tests that the next code snippet must pass. This proved to be a huge aid for a pipelined data translation system such as this one. Any unexpected behavior could be narrowed down with a high degree of confidence to recently added code, not a cascade of effects from older code. One notable drawback to test-driven

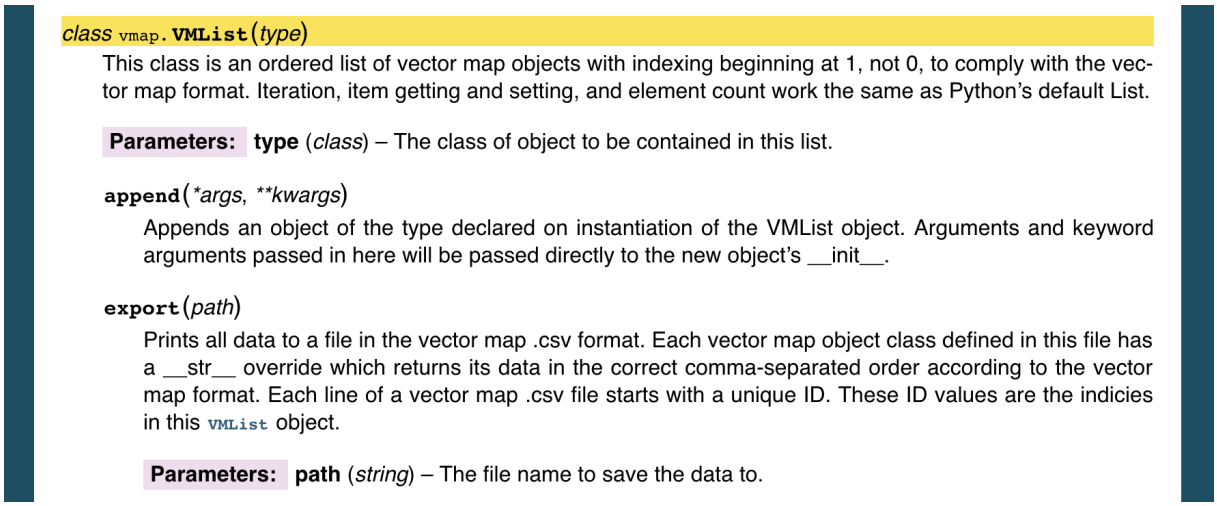
development in this project was that intial versions of code were typically done sub-optimally, probably because only black box unit tests were used.

The choice of XP as a development methodology influenced our decision to use Python instead of MATLAB. Python's third-party packages offer all the same functionality as MATLAB, plus a wider selection of testing and documentation packages. The *pytest* package was chosen over the *unittest* package found in the standard library. *pytest* requires significantly less set-up and tear-down code to achieve the same outcome as *unittest*. The online Python community largely seemed to agree that *unittest* was designed for programmers familiar with JUnit and preferred to keep the same formalities (i.e., it uses camel case naming conventions which is not standard in Python).

Summary. The activities in XP allowed the programmers in this project to quickly converge on well-justified design patterns for the software architecture. In the weekly stand-up meetings, the code segments written with the goal of passing unit tests were reviewed and adapted to conventional design patterns with agreed-upon interfaces. The iterator design pattern was used extensively in both the road generation and vector mapping subsystems in order to keep their complicated internal operation opaque to the rest of the system. The facade design pattern was used to isolate the vector mapping subsystem altogether, with only two simple interfaces to generate and link the entire map. Git version control was used throughout development, and the more modularized the code became, the easier it became for the programmers to collaborate without interfering in each others work.

3.1.3 Documentation

Concise documentation was absolutely necessary for the vector mapping code, as the stakeholder intended to expand its functionality over time and ultimately make it available to the public. Three options for Python were considered: *Doxygen*, *pdoc* and *Sphinx*. All of these support automatic HTML and LaTeX documentation generation from comment strings in Python code, with *pdoc* offering by far the simplest interface. *Doxygen* is not Python-exclusive, and it has the most complicated interface, although it is an industry standard for languages like C++ and has been around a long time. Ultimately, *Sphinx* was the best choice because Python itself uses it for documentation, so the resulting HTML docs will look familiar and accessible to any Python programmer (see Figure 3). Only public functions are documented in the HTML docs. Other variables, classes and methods are documented with ordinary in-code comments. This way, the information relevant for users of the sofware subsystems is kept separate from those who want to edit them and add functionality.



class vmap.**VMList**(*type*)

This class is an ordered list of vector map objects with indexing beginning at 1, not 0, to comply with the vector map format. Iteration, item getting and setting, and element count work the same as Python's default List.

Parameters: **type** (*class*) – The class of object to be contained in this list.

append(**args*, ***kwargs*)

Appends an object of the type declared on instantiation of the VMList object. Arguments and keyword arguments passed in here will be passed directly to the new object's __init__.

export(*path*)

Prints all data to a file in the vector map .csv format. Each vector map object class defined in this file has a __str__ override which returns its data in the correct comma-separated order according to the vector map format. Each line of a vector map .csv file starts with a unique ID. These ID values are the indicies in this VMList object.

Parameters: **path** (*string*) – The file name to save the data to.

Figure 3: Sample of the HTML documentation generated automatically from comment strings in the Python code using *Sphinx*. This class uses the abstract factory design pattern: objects are created using the same generic interface regardless of their exact type. This class also implements the iterator design pattern, but that code is implemented with private override functions, so it does not appear in the HTML docs.

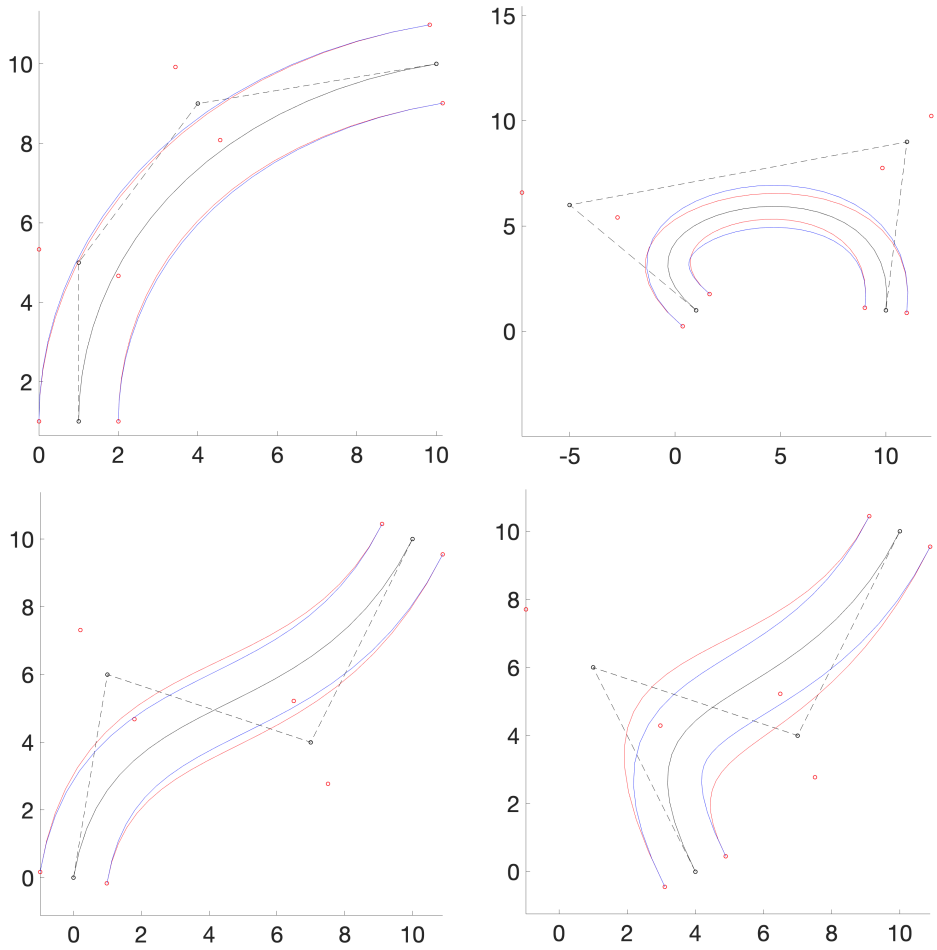


Figure 4: Comparison of two methods for approximating parallel Bezier curves. In black: the progenitor curve with its defining control points shown as circles connected by dashed lines. In red: approximation using the Tiller-Hanson method, with the associated control points shown as circles. In blue: approximation using a purely numerical method. This method was chosen for the vector map software.

4 Implementation

4.1 Vector Maps

4.1.1 Coordinate Generation

PreScan uses three basic road types: *StraightRoad* (implemented with a linear equation), *BendRoad* (implemented with a semi circle) and *BezierRoad* (implemented with a cubic Bezier curve). This much could be ascertained from the names of the road segments and their attributes in the configuration file. For the vector map, coordinates no more than 1 meter apart would need to be computed for the center of the roads, the center of the driving lanes, and the edges of the roads. The configuration file *only* defines the center line curves, so algorithms for finding parallel curves for each equation based on the lane width was the next step.

Computing parallel linear and circular curves is trivial, because the resulting curves are also linear or circular. However, a curve parallel to a Bezier curve is not another Bezier curve. To the authors' knowledge, only approximation methods exist for solving this problem. Two approximation methods were compared, which is illustrated in Figure 4. The first attempt was an implementation of the Tiller-Hanson method, which involves shifting the control points defining the progenitor curve. The accuracy of the approximation is dependent on the original arrangement of control points. Errors using Tiller-Hanson method could easily be on the scale of meters, which is unacceptable for a driving simulation. Because the vector map requires discrete coordinate values, an equation for a parallel curve was only a convenience. Therefore, a numerical approach was proposed which would guarantee accuracy (see Algorithm 1).

Algorithm 1 Compute the nearest point $\mathbf{Q}$ on the curve parallel to the Bezier curve defined by the control points $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3$, evaluated at t, at a distance d from the progenitor curve.

Require: $0 \leq t \leq 1$ and $\mathbf{P}_n = (P_{xn}, P_{yn})$ and $d \geq 0$

$\mathbf{B} \leftarrow (1-t)^3\mathbf{P}_0 + 3(1-t)^2 t\mathbf{P}_1 + 3(1-t)t^2\mathbf{P}_2 + t^3\mathbf{P}_3$

$d\mathbf{B} \leftarrow 3(1-t)^2(\mathbf{P}_1 - \mathbf{P}_0) + 6(1-t)t(\mathbf{P}_2 - \mathbf{P}_1) + 3t^2(\mathbf{P}_3 - \mathbf{P}_2)$

$\theta \leftarrow \arctan(dB_x,\ dB_y)$

$\mathbf{Q} \leftarrow (B_x + d \times \sin\theta,\ B_y - d \times \cos\theta)$

4.1.2 Software Design Patterns

5 Verification and Validation

5.1 Vector Maps

6 Results

7 Discussions and Conclusion

8 Future Work

References