VIRTUAL AUTONOMOUS SYSTEM TRAINING VAST (ENVIRONMENT)

Germaine Badio
Department of Modeling, Simulation, & Visualization Engineering
Old Dominion University
gbadi001@odu.edu

Elise Feldt
Department of Modeling, Simulation, & Visualization Engineering
Old Dominion University
efeld002@odu.edu

Zhaohui Hu
Department of Modeling, Simulation, & Visualization Engineering
Old Dominion University
zxxhu002@odu.edu

Peter Marchione
Department of Modeling, Simulation, & Visualization Engineering
Old Dominion University
pmarc002@odu.edu

Christine Odenwald
Department of Modeling, Simulation, & Visualization Engineering
Old Dominion University
coden001@odu.edu

Jaron Stevenson
Department of Modeling, Simulation, & Visualization Engineering
Old Dominion University
jstev003@odu.edu

Travis Sullivan
Department of Modeling, Simulation, & Visualization Engineering
Old Dominion University
tsull003@odu.edu

ABSTRACT
Autonomous Vehicles (AVs) have enormous potential to be the future for public transportation, delivery services, military, and other applications. Lockheed Martin Corporation (LMC) sees this enormous potential and has developed autonomous systems for air, land, and water vehicles. These AVs require testing by introducing them to different scenarios that a vehicle could encounter. The Virtual Autonomous System Training (VAST) has been developed to provide such a test environment.
Keywords:
AV testing environment, navigation simulation, collision detection, scenario object

TABLE OF CONTENTS

1 INTRODUCTION

Autonomy also delivers significant value across a diverse array of global markets. Both enabling technologies and commercial applications are advancing rapidly in response to market opportunities. Autonomy is becoming a ubiquitous enabling capability for products spanning a spectrum from expert advisory systems to autonomous vehicles. Autonomous Vehicles (AVs) have the potential to be the future for public transportation, delivery services, military, and other applications. As part of system development, many industries, including the automobile industry, make substantial use of modeling and simulation to help understand system performance. The issue of trust is core to DoD's success in broader adoption of autonomy. But these AVs require testing by introducing them to different scenarios that a vehicle could encounter so humans can determine if is operating reliably and within its envelope of competence.

The Defense Science Board has concluded that autonomy has the potential to deliver substantial operational value across a diverse array of vital DoD missions. But the DoD must accelerate its exploitation of autonomy to realize the potential value and exploit its operational benefits. [1] At Lockheed Martin Corporation (LMC), the Corporate Engineering, Technology & Operations (CETO) is leading a corporate-wide effort to visualize and understand the impact and implementation approaches for AV technologies. LMC must verify and validate the potential operational value of land-based AVs by running varies scenarios to ensure the systems are safe, reliable while contributing to mission success. To rapidly test land-based AV concepts, LMs but needs to build a simulation system that allows developers and user to design experiments, test, the system and visualize the scenarios and outcomes in a cost-effective manner.

The Virtual Autonomous System Testbed (VAST) conception will build on existing research in land navigation simulations, as well as integrate the design, development, and testing of essential software features necessary to the system implementation. It is within the scope of VAST to parse the sensor inputs and movement outputs of AVs. The proof-of-concept was determined to be for a land-based autonomous vehicle due to the prevalence of autonomous cars in the news, and current research on development of road AVs.

As demonstrated in Figure 1 below, the overall system is comprised of three major software modules: the navigation simulation(s), the AV/software, and VAST. VAST is comprised of the replication run and the User Interface (UI) with its data and visualization components. User-defined parameters and metrics will be entered into the User Interface, and the system will generate a visualization based on these parameters and metrics. Each auxiliary module (simulation, AV / software, and UI) will send and receive information to and from the AV Testbed.

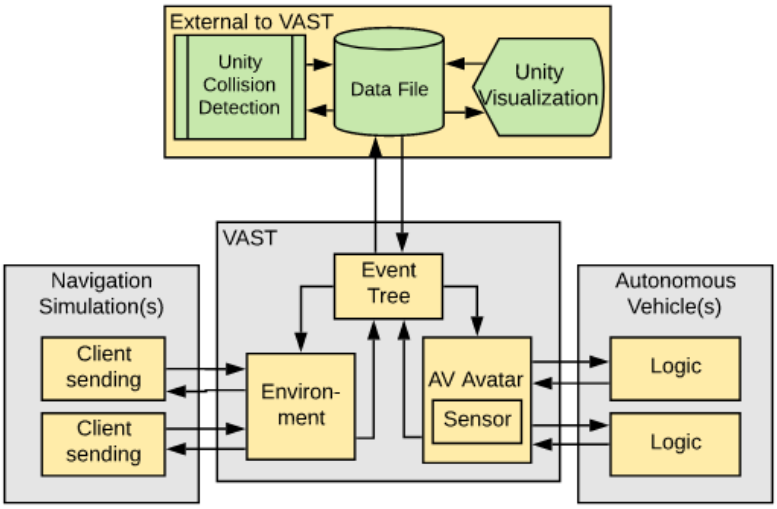

Figure 1: Testbed System Architecture

Requirements Overview

This section lays out the requirements of the AV Testbed in Table 1. The requirements are broken up into three categories: the AV Testbed Process (what the internal testbed components are and how they interact), AV Testbed Inputs (from the user interface, the navigation simulation(s), and the AV /software), and AV Testbed Outputs (to the user interface, the navigation simulation(s), and the AV /oftware). Each requirement also has associated acceptance criteria, as outlined below.

Table 1: System Requirements

Requirement	Acceptance Criteria
AV Testbed Process	
AV Testbed shall accurately represent data input to clients.	With sensor parameters provided, the simulation must be able to understand the AV's decision. This decision must be representative of the AV's actual decision.
AV Testbed shall visualize the environment accurately.	The virtual environment Visualization must represent AVs and other obstacles' position and geometry accurately.
AV Testbed Inputs	
AV Testbed shall be remained general enough to handle multiple sensor and vehicle types that can be inputted by the user. These	The user must be able to add new sensor types with different parameters and functionality. The sensor parameters must provide the simulation with enough

sensor and vehicle types shall have acceptable and realistic parameters.	information to understand how the sensor will act including what types of data are associated with it.
AV Testbed shall receive and store scene simulation information.	The simulation information must be abstracted to handle multiple types of simulations.
AV Testbed Outputs	
AV Testbed shall accurately state the output made by the AV.	The geometric output may include information about the state of the environment surroundings, the AV, the sensor readings, and the resulting output of the vehicle.
AV Testbed shall evaluate whether a failure/operational mission failure has occurred and output its result.	A failure is defined as any software issue that results in unpredicted behavior. An operational mission failure (OMF) is a failure that prevents the entire system from being used (e.g. crash).
AV Testbed shall report relevant performance measures	Performance measures include quantitative and qualitative information regarding the overall performance of the AV. Relevant measures are described in Section 2.4.

2 SOLUTION DESIGN

2.1 Testbed Components

The VAST Testbed refers to the software executable at the center of the VAST system seen in Figure 1. When the Tester is ready to begin a scenario, she will configure the VAST Testbed to control the start and run time of the clients (AV navigation logic and the navigation simulation). Modules in the VAST Testbed will receive known port names and variable configurations for communication with AV logic and navigation simulation. The data will be intercepted for scenario analysis. Modules in the VAST Testbed will perform data relay, data storage, and analysis calculations. Further modules will interface with the VAST Testbed user.

The data flow between components is illustrated in Figure 2. Each component block is a module of functionality written in programming code. An arrow describes the movement of data from source to destination.

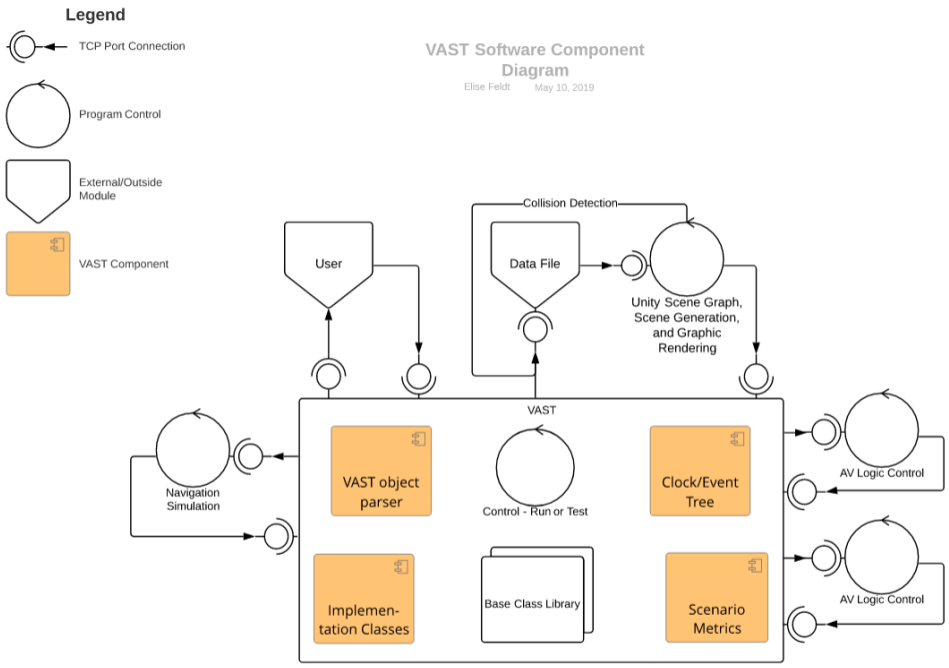

Figure 2: VAST Component Diagram

Organization

The configuration wizard classes will perform complex routines involving the naming of scenario objects, the configuration of communication ports, and initialization variables needed from the clients to run VAST. During initialization the user will also be prompted to select metrics and the location of the output database.

The scenario objects will source from the Object Class Library to create objects with open port connections and data structures to complement the translation of incoming client data to outgoing client data. Events generated by port activity will disseminate to the associated scenario objects and the event tree.

As the development of the VAST program has evolved, the importance of the types of data flow has emerged as seen in Figure 3. Every arrow in the diagram below indicates part of the VAST program that was written to convey meaningful information from one component to another. All the data is coming from the testing of AVs.

The initial black arrow going from the user to the Configuration Wizard represents the programming of the QT graphical user interface and the initial efforts the user will undergo to set up their VAST system components. Subsequent orange arrows symbolize the programming inside of VAST that will use the configuration specifics to initialize components and TCP connections.

This initialization is a form of automation for the VAST user, allowing them to access VAST features from a single control point once the components have been decided.

Internal to the VAST program, data will flow via updates (blue) and events (purple), with the EventTree and its Clock time as a hub that receives and organizes data flow by time stamp and sends it to the database file. An update would be characterized by any information modified from external sources that changes attributes of the sending component. A sendEvent would be characterized by information the EventTree has organized and is ready to distribute to all the components in the system distribute to all relevant components.

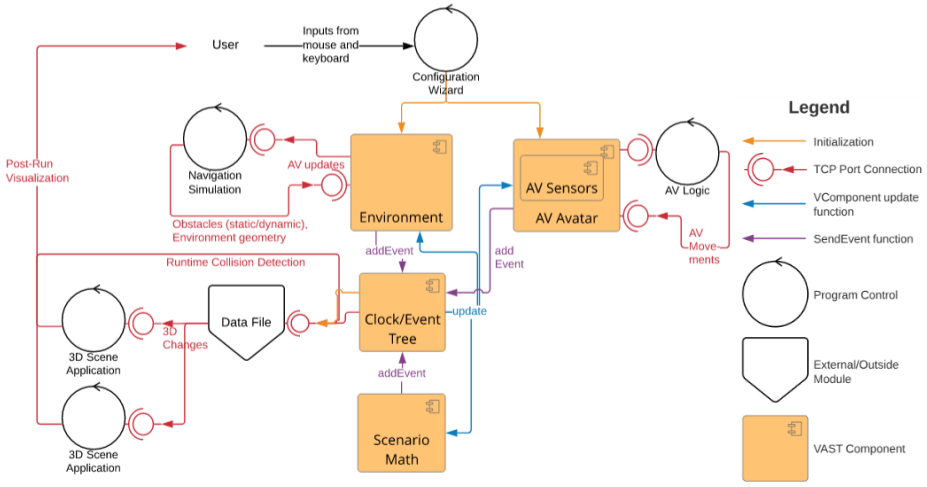

Figure 3: VAST Data Flow Diagram

Extension

It is the aim of the VAST product to make extensible classes for the AV tester to create their own Environment, Obstacle, AV, and Sensor classes. Extensibility of these base classes will allow the user to create complex AV sensor types and more elaborate environments in which to test the AV.

There are currently no designs for a user interface for visual programming of extended Scenario Object classes.

2.2 Software Design

The VAST program will be written with Object Oriented Design, as referenced in the Testbed Components section above. The source code for VAST is stored in a GitHub repository. A user guide will be generated for VAST developer/user reference. The user guide should be developed as VAST Scenario Objects are implemented and functionality is expanded. VAST's initial Scenario Objects Library class has been included in Appendix A.

The abstract classes that are used in the current prototype include the Scenario-Metric, AV, Sensor, Environment, and the Their parent class, the VComponent. All these classes allow the user to set up different extended specialization classes that can be used anywhere that the original class is used. This allows the user to create metrics that they would like to track as well as add more autonomous vehicles under the AV class. It is the aim of the VAST product to make extensible classes for the AV tester to create their own Environment, Obstacle, AV, and Sensor classes. Extensible base classes will allow the user to create tests with complex AV sensor types and more elaborate environments.

Time, Events, and Synchronization

A simulation cannot exist without some form of time measurement. A simulation can use either continuous or discrete time progression. However, an AV is assumed to operate in the real world with continuous time only. Yet both the AV and the navigation simulation must perform calculations with respect to state changes over time. Metrics and calculations are more readily accessible with a constant unit of time by which they measure those state changes. Because of this higher relevance for regular intervals of calculations, and the desire to update a visualization, the choice was made to follow a discretized method of time progression.

In order to bring together the state changes of these two client applications, VAST will adopt its own discretized event recording. The advancement of the clock time will be driven by VAST.

It is important to the virtual environment testing to run multiple replications and multiple scenarios as fast as possible, so the AV system logic may likely be required to take in and output a timestamp of its own, or implementation of the AV may be instructed to ask the EventTree for the current time.

The VAST Tester will ideally choose a minimum time slice with which the EventTree will prioritize incoming Events. Events that fall between time slices will be grouped and considered a single event, with priority given to the latest added event.

Database

The database is set up based using SQLite3. The current database file uses a SQLite library. If a database does not exist, one will be created, and a database object will be returned. The Unity module will query data from the open database to generate scene graphics and perform collision detection during the run. The collision detection in the scene graph is during run time and that data will be published to the database file. The Scenario Math classes will perform live metric calculations and general physics calculations at each Scenario Object update, and the calculation will be sent to the database file. The Event Tree/Clock class will perform the function of collecting and maintaining AV and Environment events and publishing them.

2.3 Proof-of-Concept

The primary method for controlling a vehicle in SUMO (Simulation of Urban MObility) would be to use the MoveToXY command [2], which would require an update at every time-step

to continue moving the vehicle. Although using MoveToXY may be feasible, the control methods in other simulation software products are far such as more intuitive.

Implementation

The current plan for implementing both simulations for use with the AV is to use OpenStreetMap [3] along with various tools that have been developed to convert OpenStreetMaps to other formats (including Unity and Unreal environments, and SUMO networks) [4], [5], [6]. This would allow for the same real-world location to be represented in both simulation products, easing the process of having consistent scenarios.

This coordination of multiple simulation products will not be without challenges, as this introduces factors such as time-synchronization between the two simulations. However, this will be an ideal way of showing the full functionality of VAST, as well as helping to ensure VAST is developed with enough flexibility to enable such feats.

Collision Detection and Post-Simulation Visualization

Another consideration is that not all navigation simulations have the capability to detect collisions, which is an integral part of training and testing an AV. To account for this, the Proof-of-Concept will also include Unity-driven collision detection.

The final component of the Proof-of-Concept involves the generation of a post-run visualization (as described later in section 3.3). This functionality allows users the opportunity to visualize a scenario after it has been run and interact with it: increase and decrease update speed, pause the simulation, rewind and fast-forward one frame at a timetoggle camera views, and increase as well as decrease run number. This is highly useful in cases where the navigation simulation does not inherently support 3D visualization.

Figure 4 : Post-Simulation Visualization

2.4 Verification for Design

Verification of VAST must be performed to ensure that it meets all project requirements as addressed in Table 2. Each requirement will have a test description, acceptance criteria for accepting that the requirement is met, and a description on how these requirements solve the problem statement. Criteria will include measures that may be thresholded or un-thresholded. The Minimum Viable Product (MVP) of VAST is considered complete when each measure of verification passes. A thresholded measure has a specific valued condition that must be met as a condition for the system to pass the test; in other words, it is a critical measure. However, an un-thresholded measure means that the value is calculated and tracked but is not required for the system to pass. Critical Operational Issues (COIs) will be addressed as well in the test plan; these include effectiveness (mission support) and suitability (reliability, maintainability, logistic supportability, and availability) determinations. This plan must be developed in accordance with the requirements and its acceptance criteria, which are described in Table 1 and will utilize design of experiments (DOEs).

Table 2 below lists the unit test procedures, associating each requirement with specific components within AV Testbed.

The collision detection section includes information about what requirement it is fulfilling, some information on what the collision detection module needs to report to the database, and descriptions of the test scenarios. The requirement that it is fulfilling is the testbed outputs requirement of evaluating whether a failure/operational mission failure has occurred and output its result. The module should report the run number, scenario number, time of the collision, and the object the AV collided with to the database. There are four test scenarios: post-simulation example to file, post-simulation example to the database, faster-than-real-time example to file, and faster-than-real-time example to database. The test scenarios with the files will be completed within the Unity editor; however, the scenarios with the database will be a Unity executable application.

The Proof-of-Concept demonstration will be used to validate that the entire system works together properly; thus, it serves as the integration test for VAST. While the focus of the test plan has been on the unit testing of the classes, there is also the integration testing to take into consideration. An integration test serves as the validation of the component interactions; the unit tasks prove the inner functionality of each component in isolation. Thus, a combination between these two tests will validate the functionality of this project.

Overall, there will be limitations to testing VAST. The major limitation is the fact that this test does not include hardware-in-the-loop, which means that this validation is purely on the software. Even if this software is fully validated, new problems are likely to arise once both hardware and software are tested in conjunction. Specifically, in defense acquisition, multiple test events are conducted. In an Operational Test (OT) report from Fiscal Year (FY) 17, 64 OTs were conducted with Director, Operational Test and Evaluation (DOT&E) oversight [7]. 34.4% OTs

conducted in FY17 discovered critical, new problems in addition to old problems [7]. 15.7% discovered only new, critical problems [7].

Additionally, there is no way a system can account for every scenario without testing how the system of systems (SOS) integrates with the system under test (SUT). This includes how the hardware interacts with the software along with other simulation types.

Table 2: Unit Test Procedures and Associated Measures

Requirement	Unit Test Procedures	Measures
Process: AV Testing Environment shall accurately represent data input to clients.	-Tester will run scenarios that require data transfer between AV testing environment and the simulation/AV. -Information regarding data accuracy and transfer rate is recorded and analyzed.	-Data format reliability -Data transfer success rate
Process: AV Testing Environment shall visualize the environment accurately.	-Generate a virtual environment based on the simulation and AV interaction. -Determine if the AV position in the AV testing environment scene graph is reliable.	-AV position and orientation reliability in visualization
Inputs: AV Testing Environment shall be abstracted to handle multiple sensor and vehicle types that can be inputted by the user with acceptable and realistic parameters.	-Tester selects multiple sensor types. -AV testing environment is expected to instantiate these sensors and associate them with the AV. -Sensor list is printed out and checked against the expected sensor listing.	-Sensor list reliability
Inputs: AV Testing Environment shall receive and store scene simulation information.	-Tester runs a scenario with a chosen simulation. -Scene information is stored internally to AV testing environment. -Stored scene information is compared to the expected scene information.	-Scene object position reliability -Scene update rate
Outputs: AV Testing Environment shall accurately state the output made by the AV.	-Tester runs multiple scenarios. -AV testing environment sends information to the AV and receives input back from the AV. -Output is compared to the expected output..	-Output reliability
Outputs: AV Testing Environment shall report	-Tester runs scenarios that force collisions between the AV and other vehicles.	-Collision detection rate

whether a failure/operational mission failure has occurred.	-AV testing environment outputs the collision information.	
Outputs: AV Testing Environment shall report relevant performance measures.	-Tester selects metrics. -AV testing environment runs a test scenario with already-known metrics. -AV testing environment generates a metrics report. -Tester compares the generated metrics with the known metrics.	-Proper number of measures are reported -Measures are correct for the scenario

2.5 Integration Testing

Integration tests serve as the validation of the component interaction while unit tasks prove the inner functionality of each component in isolation. The combination of unit and integration testing will validate the functionality of VAST.

The proof-of-concept will serve as the integration test for VAST. It demonstrates the interaction between simulation, the autonomous vehicle, and the test bed itself. This test focus on the socket connection and validating the information within the data packets that are sent to and from various components. Additionally, the integration test will demonstrate that collision detection is working properly. This includes being detected by the correct module and reported to the database with the proper information

3 VAST TESTBED USER EXPERIENCE (UX) AND OUTPUT

3.1 System Flow

One of the most important steps in the software process is the basic user experience (UX), which for this project focuses around initialization and configuration, which consists of its own "wizard," laid out in the configuration wizard UX. This process also accounts for a post-run view that will display metrics selected in the initialization step. This view will include aggregated metrics such as average speed, maximum speed, minimum speed, etc. Multiple replications can be selected, all replications will be run before this screen is displayed, and metrics will be displayed per replication. VAST will generate an output database (.db) file. Within the run summary screen, the user will have several options as to how to continue. The simulation can be re-run, the user can go back to the initialization step, an interactive visualization can be generated based on the data from the run or the program can be exited.

The configuration wizard allows a user to select configurations for the simulation and AV, determine important metrics, select number of replications, choose sensors to use, and determine the location for any output file. The most important part of the configuration wizard is the behind-the-scenes initialization of the simulation and AV that occurs based on the information provided

by the user in configuration files. This allows the user to determine things such as how many vehicles should be present in the simulation, what AV logic to use, starting position of the AV, etc.

3.2 Metrics

Performance metrics are necessary because the user can choose to perform simulations faster than real time. It would be infeasible for a user to only visually determine the information they need in such a scenario.

Table 3 includes performance metrics chosen for their suitability to be implemented into VAST by default. "By default" means that the metric is applicable to most AVs and scenarios. Many of the metrics below would make excellent measures of AV performance, and the Tester will be able to extend the ScenarioMetric base class in a desired implementation.

Table 3: Viable Metrics

Minimum distance	The distance between the autonomous vehicle and any object. The object can be dynamic – like another vehicle in the simulation – or static – such as a building that a drone must avoid.	Not Included
Maximum acceleration /deceleration	Units may be selected by the user (feet, meters, etc.). They measure, respectively, the maximum rate of increase and decrease in the AV's velocity.	Included
Miles per collision	Viable if the simulation does not end after the AV makes a collision. If this is the case, VAST will keep track of how many collisions occur and how many miles occur between them.	Not Included
XYZ location, average speed/ acceleration	Since we are keeping track of X, Y, and Z location by default, we can calculate the average speed and average acceleration from this information.	Included
Congestion	A metric that represents how many dynamic obstacles are around the AV within a given space. This will be useful in determining if the AV is generating traffic due to the way it behaves.	Not Included
Worst time gap between dynamic obstacles (front/back)	Represents the smallest distance "in time" between two ground vehicles. For example, if driving on the highway at 55 mph and the car in front of the AV passes a pole, and it takes 3 second for the AV to pass that pole, then the time gap between them is 3 seconds. The way this is different from following distance is that if both vehicles in the example are driving at 25 mph, then the distance	Not Included

	between them can be smaller, but the time gap between them might still be 3 seconds.	
Stop frequency	How often the vehicle must stop along its route. This may be useful to determine if the AV is making more stops than necessary.	Not Included

Additionally, a standardized testing arena is something the Intelligent System Division at the National Institute of Standards and Technology is proposing in order to facilitate standard performance metrics. These Autonomous Road Driving Arenas (ARDA) would ideally promote the creation of AVs by providing a standard testing method and "lifting the standards of development within the community by providing comprehensive data sets, publicly accessible arenas, and competitions." It would make sense that a virtual replication of said arenas could be equally as beneficial for testing an AV's decision making during early development [8].

One crucial aspect related to performance metrics is the feasibility of this testing arena, although not within the scope of this project, one crucial aspect to determine that feasibility. In the case of a car AV, it would be impossible to measure the aggressiveness of the onboard AI if there are no other vehicles for the car to interact with, nor would it be possible to determine how well it handles the aggressiveness of other vehicles on the road. It is also important to have a repeatable scenario in order to see the results of adjustments to the AV logic. The feasibility of the scenario in turn lends to the feasibility of the AV performance metrics.

A feasible testing arena for any AV type would have features that can adequately allow the AV to perform a feature test. For a car AV's camera feature, for example a feasible testing arena must include objects visible in varying degrees of the color spectrum and in different lighting conditions that is picked up by that camera. Objects must move on and off screen and move at and faster than the camera frame rate. Objects must range from smaller than a pixel size to larger than the camera frame size. In this way, the metrics developed for the camera feature testing will confidently report the failure parameters of the AV's camera sensor.

3.3 Visualization

The Proof-of-Concept will have an application (based in Unity) to be optionally used as a plug-in by the VAST Tester. The purpose of the Unity-based visualization is to show the user the combined scenario of the simulation and the AV. The overall logic of the visualization as seen in Figure 4 below begins by reading in an output file from SUMO that contains second-by-second position, rotation, and velocity data for each vehicle in the simulation. The number of vehicles is determined based on this file, and the vehicles are instantiated into the scene after runtime. The user of the visualization selects playback speed, and the Unity program updates all vehicle information in the viewing window (position and rotation) accordingly.

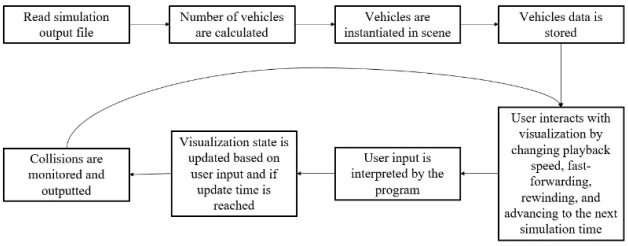

Figure 4: Visualization Logic

In the demo version of this visualization, there are multiple public variables (a reference to vehicle meshes/camera/text objects and time interval between simulation updates) and private variables (a reference to the stream reader, current line of the data file, number of vehicles, vehicle list, current visualization time, space pressed, and enter pressed). The program initializes the update time, reads through the entire data file, interprets the first line of the data file to determine the categories and number of vehicles, and declares arrays and list.

Additionally, the update function is called once per frame, checks multiple conditions, and updates certain state conditions. The camera is updated based on the position and rotation of the first vehicle. The user can press the spacebar, the enter button, and the arrow keys to interact with the visualization.

The input received is interpreted by the program and is used to update all user interface information. These include five text object references: time text, car following text, position text, rotation text, and updates time text (time between visualization update). Figure 5 below demonstrates an example scenario where the bus collides with a car. The simulation time, the name of the vehicle being followed, the world position, the rotation of the vehicle, and the time between visualization update in seconds are all included in the left panel. Additionally, the "paused" visual feedback, graphics/audio statistics, and user input instructions are included.

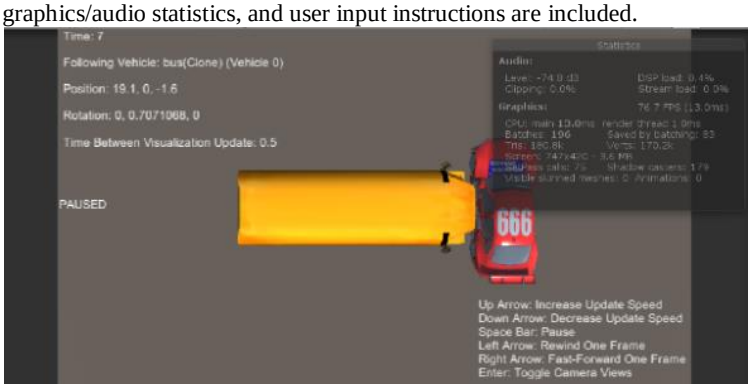

Figure 5: Visualization Example Scenario

Additionally, collision detection information is outputted using the "OnCollisionEnter" built-in Unity function [9]. To increase performance of the visualization and collision detection, a simple box collider is be used to encompass all obstacles in the scene, while a mesh collider is used to encompass the AV. The major difference between these two colliders is the mesh collider is shape-specific [10]. Collision detection is controlled by the physics system in the Unity program [11]. The layer collision matrix defines what object types can collide with other object types. Additionally, Unity allows for two major broad-phase types: sweep/prune and multibox pruning. Continuous collision detection is also an option, which ensures that objects with a high velocity do not simply pass through other objects without the physics system recognizing the collision [12]. While it may reduce the performance of the simulation, it is necessary for certain use cases (e.g. high-speed vehicles).

3.4 VAST Configuration Wizard

The Configuration wizard is divided into three sections: VAST, AV, and Environment. In the VAST tab, the user has the ability to select the location and the name of the configuration file. The user can also decide when the visualization will occur either during or after the simulation. The simulation time can be modified from faster than real time to real time. The number of repetitions, max time, and time steps can be set as well. The user is allowed to add and select the metric being used and set the list seed. Figure 6 shows the main window of the configuration wizard and the VAST tab.

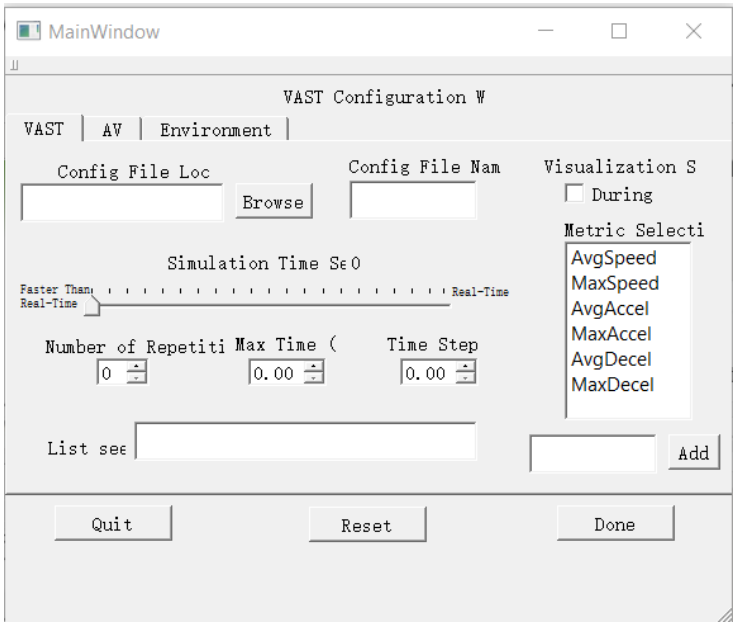

Figure 6: Configuration Wizard (VAST Tab)

Figure 7 shows the AV Tab of the configuration wizards and the options that are presented to the users. Within this tab, the user can add, delete, and edit an AV program file to VAST.

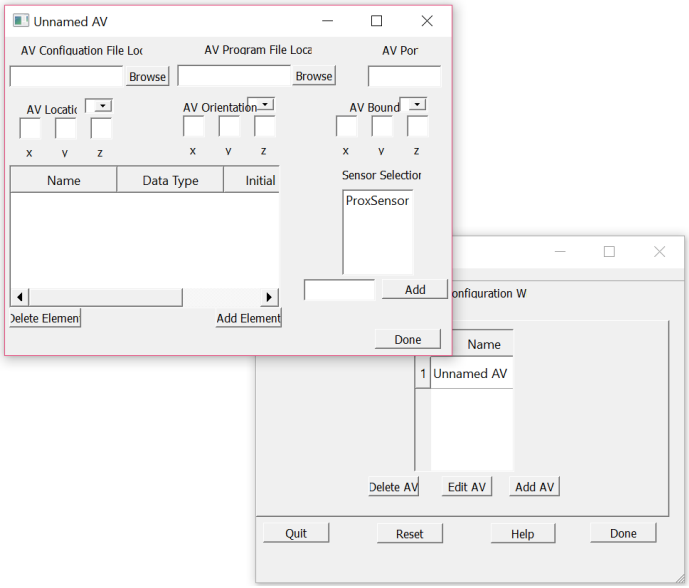

Figure 7: Configuration Wizard (AV Tab)

The Environment tab shown in Figure 8 allows the user to select the locations of the environment configuration and program file. The environment port can be defined by the bounds of the environment (the x,y,z position and units of measurements). The user can also add and delete elements as well as choose the name, the data type and the initial value of the element.

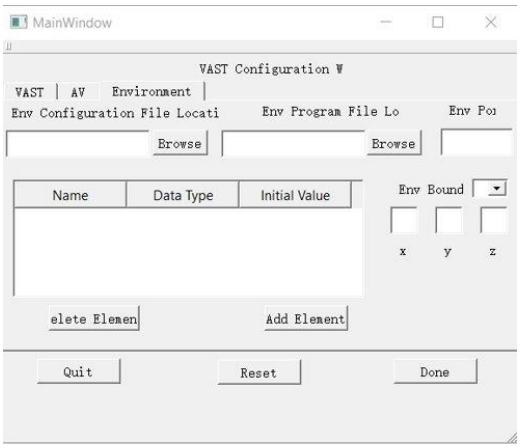

Figure 8: Configuration Wizard (Environment Tab)

4 CONCLUSION

The project to create a virtual autonomous system training environment was accomplished to the point of proof of concept in the span of two semesters. The team of seven undergraduate Capstone seniors developed a code base that managed and observed the correspondence between two separate and distinct processes: the autonomous logic, as represented by a script, and SUMO environment. These two processes were able to communicate using the VAST system architecture, and the event of collisions was successfully interpreted as a failure mode. At this time, the development of VAST is still in its infancy. There are features possible with the extensible design that the Capstone team was unable to implement by the May delivery date, but which were thoroughly discussed with the Lockheed Martin customer. The team hopes that the extensible design will be utilized for the rapid, inexpensive design of experiments planned by Lockheed Martin, so that the development of all types of autonomous vehicles can be trained toward exhaustive confidence in autonomous systems.

5 GLOSSARY

Term	*Explanation*
AV Performance Metric ("metrics")	A calculation of AV or scenario data in a specified formula that lends meaningful analysis to the Testers of the AV about its performance in any scenario.
Packet	A unit of encapsulated data that can be sent and received over a network and unpacked to become meaningful information for the recipient.
Scenario	A combination of 3D environment and objects and other potentially non-3D elements that together create a distinctive set of parameters for an AV to navigate.
Testing Arena	The precise combination of scenario elements that must be present to test the performance of the AV.
Tester	The future user of VAST, a developer of AV logic, or even potentially a developer of navigation simulations.
Operational mission failure	The failure that occurs during a simulation run, signifying the conditions for successful measures have not been met, but that the run can continue otherwise.
VComponent	Vehicle component, which include the name of vehicles, type of vehicles and other information of vehicles.
Event Tree	An organization of events in a simulated software program. Named "tree" because of the association between unique events.

Critical Operational Issues	The failure that occurs during a simulation run, signifying the conditions for successful measures have not been met and the run cannot continue due to a failure that cannot be overcome.
Sweep/prune	A broad phase algorithm for collision detection that checks a limited number of solid space objects for intersection.
Multibox pruning	A broad phase algorithm for collision detection that pairs solid objects and then limits the number that need to be checked for intersection.

6 ACRONYMS

Commented [MP1]: I double checked this paper for acronyms. Here's what's missing. https://i.imgur.com/ndxq82d.png

3D: Three-dimensional
ARDA: Autonomous Road Driving Arenas
AV: Autonomous Vehicle
CETO: Corporate Engineering, Technology & Operations
COIs: Critical Operational Issues
CPU: Central processing unit
DOEs: Design of experiments
DOT&E: Director, Operational Test and Evaluation
FY: Fiscal Year
LMC: Lockheed Martin Corporation
MDI: Mission Development and Integration
MVP: Minimum Viable Product
OMF: Operational mission failure
OT: Operational test
SOS: System of systems
SUMO: Simulation of Urban MObility
SUT: System under test
TCP: Transmission Control Protocol
UI: User Interface
UML: Unified Modeling Language
UX: User Experience
VAST: Virtual Autonomous System Testbed

7 REFERENCES

[1] R. A. David and P. Nielsen, "Defense Science Board Summer Study on Autonomy," June 2016. [Online].

[2] "TraCI/Change Vehicle State," 28 November 2018. [Online]. Available: http://sumo.dlr.de/userdoc/TraCI/Change_Vehicle_State.html. [Accessed 6 February 2019].

[3] "OpenStreetMap," [Online]. Available: https://www.openstreetmap.org/. [Accessed 6 February 2019].

[4] M. Fricker, "Street Map Plugin for UE4," 18 June 2018. [Online]. Available: https://github.com/ue4plugins/StreetMap. [Accessed 6 February 2019].

[5] Mapbox, "Maps SDK for Unity," [Online]. Available: https://docs.mapbox.com/unity/maps/overview/. [Accessed 6 February 2019].

[6] "Networks/Import/OpenStreetMap," 26 January 2019. [Online]. Available: http://sumo.dlr.de/wiki/Networks/Import/OpenStreetMap. [Accessed 4 February 2019].

[7] "Problem Discovery Affecting OT&E," [Online]. Available: https://www.dote.osd.mil/pub/reports/fy2017/pdf/other/2017problemdiscovery.pdf.

[8] C. Scrapper, S. Balakirsky and B. Weiss, "Autonomous Road Driving Arenas for Performance Evaluation," 24 August 2004. [Online]. Available: https://ws680.nist.gov/publication/get_pdf.cfm?pub_id=822519. [Accessed 6 February 2019].

[9] Unity Technologies, "Collision," 17 January 2019. [Online]. Available: https://docs.unity3d.com/ScriptReference/Collision.html. [Accessed 6 February 2019].

[10] Unity Technologies, "Colliders," 17 January 2019. [Online]. Available: https://docs.unity3d.com/Manual/CollidersOverview.html. [Accessed 6 February 2019].

[11] Unity Technologies, "Physics," 17 January 2019. [Online]. Available: https://docs.unity3d.com/Manual/class-PhysicsManager.html. [Accessed 6 February 2019].

[12] Unity Technologies, "Continuous Collision Detection (CCD)," 17 January 2019. [Online]. Available: https://docs.unity3d.com/Manual/ContinuousCollisionDetection.html. [Accessed 6 February 2019].

[13] "Database (SQLite) Setup for Unity," 8 July 2014. [Online]. Available: https://answers.unity.com/questions/743400/database-sqlite-setup-for-unity.html. [Accessed 6 February 2019].

[14] S. Cope, "TCP/IP Ports and Sockets Explained," 6 July 2018. [Online]. Available: http://www.steves-internet-guide.com/tcpip-ports-sockets/. [Accessed 6 February 2019].

[15] C. R. Rickarby, "Autonomous Driving Platform Performance Analysis," 2017. [Online]. Available: https://scholars.unh.edu/cgi/viewcontent.cgi?article=1344&context=honors. [Accessed 9 February 2019].

[16] W. G. Phillip J Durst, "Levels of Autonomy and Autonomous System Performance Assessment for Intelligent Unmanned Systems," April 2014. [Online]. Available: https://apps.dtic.mil/dtic/tr/fulltext/u2/a601656.pdf. [Accessed 6 February 2019].

[17] Nvidia, "Nvidia Pro Pipeline," [Online]. Available: https://developer.nvidia.com/nvidia-pro-pipeline. [Accessed 6 February 2019].

[18] S. Tamilarasan, D. E. Jung and L. Guvenc, "Drive Scenario Generation Based on Metrics for Evaluating an Autonomous Vehicle Controller," in *WCX World Congress Experience*, 2018.

[19] D. Stanek, E. Huang, R. T. Milam and Y. A. Wang, "Measuring Autonomous Vehicle Impacts on Congested Networks Using Simulation," in *Transportation Research Board Annual Meeting*, Washington, DC, 2017.

[20] "Welcome to AirSim," [Online]. Available: https://microsoft.github.io/AirSim/. [Accessed 6 February 2019].

[21] Unity, [Online]. Available: https://unity3d.com/. [Accessed 6 February 2019].

[22] Instituate of Transportation Systems, "SUMO - Simulation of Urban MObility," [Online]. Available: https://www.dlr.de/ts/en/desktopdefault.aspx/tabid-9883/16931_read-41000/. [Accessed 7 February 2019].

[23] A. Fox, "The Difference Between a CPU and a GPU," Make Tech Easier, 31 January 2017. [Online]. Available: https://www.maketecheasier.com/difference-between-cpu-and-gpu/. [Accessed 6 February 2019].