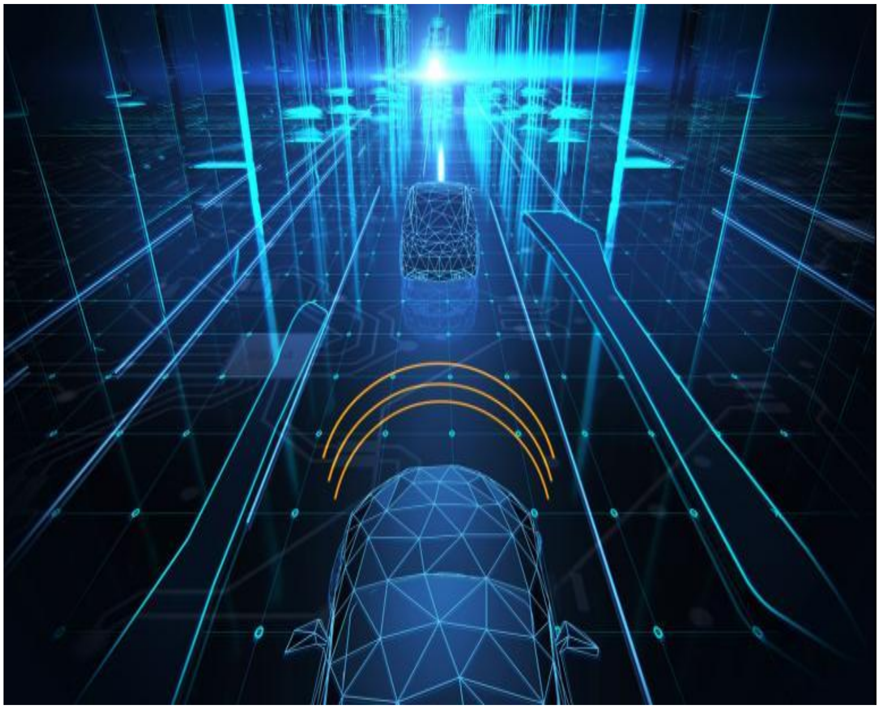
User Guide for

Virtual Autonomous System Training (VAST) Environment



Contents

1 Introduction

1.1 Purpose

VAST is a virtual autonomous system training simulation that is used to test and validate autonomous vehicle algorithms. Autonomous vehicles need to be tested in numerous different scenarios for the purpose of analyzing how the autonomous system responds to different situations and environments. The purpose of this guide is to help first time users navigate the different aspects of Virtual Autonomous System Training (VAST). VAST contains high level components such as VAST's executable (Source code for VAST), one or more navigation simulation programs, one or more autonomous vehicle (AV) logics, a visualization component, and a collision detection component. For the proof-of-concept presented by Old Dominion University's 2019 Capstone team, the Simulation for Urban MObility (SUMO) was utilized as the navigation simulation program, a Python script was chosen to stand in for up to two AVs, and two independent Unity applications were developed for each: the visualization component, and collision detection components.

1.2 Proof-of-Concept Software

1.2.1 Unity

Commented [OCE1]: Collision detection and post-sim visualization

Unity contains multiple features and tool to help with fast editing and iteration in a person's development cycle. Unity has an All-in-one editor, this feature is available on Windows, Mac, and Linux. This feature also contains a vast range of tools for the sole purpose of designing immersive environments. Unity has intuitive user interface as well as physics engines, and custom tools. It supports 2D and 3D development with other tools. It features and API for database connection and scene input. It also features meshes that can be created from an object or scene geometry, in addition to customizable broad-phase collision detection. For more information on Unity and its features, please visit their site [1]. With the features mentioned earlier in this section, Unity was chosen for visualization of the scene involving the AV and the simulation in this current of this project. Unity is used to show the collision detection which is controlled by the physics system that is available in Unity.

1.2.2 **S**imulation of **U**rban **Mo**bility (SUMO)

SUMO is an open source, microscopic, multi-modal traffic simulation and it gives its user the ability to stimulate traffic that contains vehicles that move through a given road network. Each vehicle moves explicitly, has its own route and individually through the network. Sumo contains a road network consisting of nodes which are represented as junctions and edges are used to connect the junctions. SUMO comes with features and tools to help the user prepare and run a traffic simulation in SUMO. Some of these features include space-continuous and time discrete vehicle movements, access to different vehicle types, multi-lane streets with lane changing, different right of way rules and traffic light, and it comes with a fast openGL graphical user interface. For more information on SUMO and its other features, please visit their site [2]. SUMO generates the information needed for the simulation which involve information for each vehicle in the environment. The vehicle information is displayed in an output file in which Unity reads.

2 VAST Graphical User Interface (GUI)

VAST graphical user interface was built using Qt and allows the user select from metrics of their choosing and the number of simulation runs. There are three tabs available on the main window and they are the VAST, AV, and Environment tabs. Each tab has their own set of options for the user to customize. Section 2.2 will be covering the different tabs and their options for the user in greater detail. It will also give location of all the option and a look at each tab of the configuration wizard.

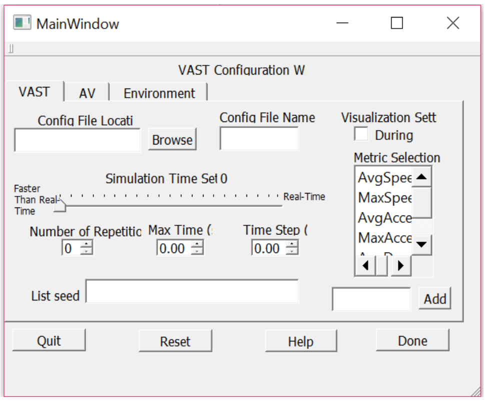
2.1 Step by step Instructions

1. Step by step Instructions
    a. User Interface
        i. Run the VAST User Interface program
        ii. Initialize Scenario
        iii. Run Scenario
        iv. Display Run Summary
        v. Re-config/Re-run/Visualization/Exit? (Before Visualization)
            1. Re-configuration: Initialize Scenario (step two) and follow previous steps
            2. Re-run: Run Scenario (step three) and follow previous steps
        vi. Visualization: Run Interactive Visualization
        vii. Re-config/Re-run/Exit? (After Visualization)

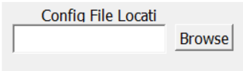
Commented [SR2]: We don't have to have this image in here since it was just a rough mockup

2.2 Configuration Wizard

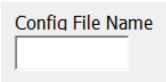
2.2.1 VAST Tab



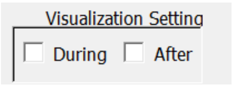
1. Choose a location for the Configuration file that will be generated (located at the top left of the main window). Click the browse button shown below, to choose a destination for the output file.



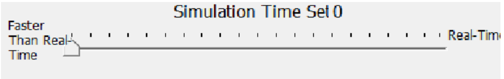
2. Choose and type the name for the Configuration file. (This is location to the right the Configuration File Location option)



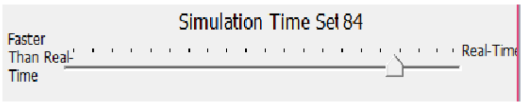
3. Choose whether to set visualization setting During or after simulation is finished. (This is located on the top right on the main window, next to the Configuration File location option)



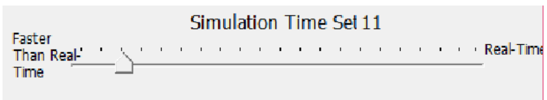
4. Choose simulation time setting: Between faster than real time (left) and real time (right). This is location below the Output File name option



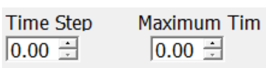
   a. Settings can range from 0 to 99
   b. By clicking on the Simulation time Set option, the arrow keys can also be used to
      i. move closer to real time (up and right keys)



c. move closer to faster than real time (down and left keys)



5. Choose the timestep and the Maximum Time Step by clicking the up and down buttons or by using the up and arrow keys (Located underneath Simulation Time Settings)

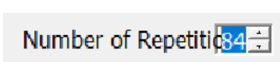


6. List Seed (Located below Simulation Time Settings and Number of Repetition Option)

7. There is an option to choose the number of repetitions which ranges from 0 to 99 (located to the right Max Time settings)

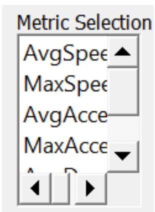
a. By clicking on the box next to Number of Repetition,

i. The up directional key can be used to increase the number of repetitions

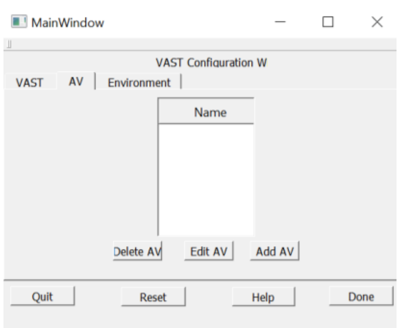


ii. The down directional key can be used to decrease the number of repetitions

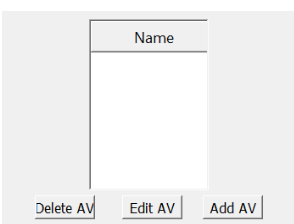
8. Choose what metrics You want to use (Located to the right Number of Repetitions)



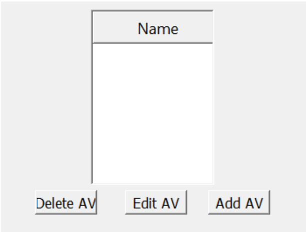
2.2.2 AV Tab



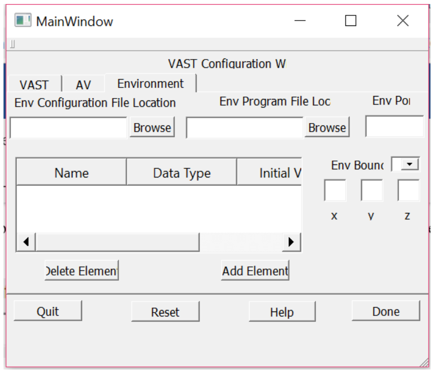
1. Option to add, edit, and delete AV



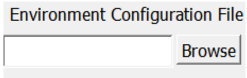
2. Option to add, edit, and delete AV



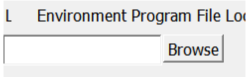
2.2.3 Environment Tab



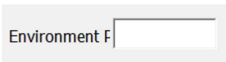
1. Select the location of the environment configuration file (Located at the top left of the main window)



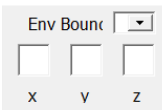
2. Select the location of the environment program file (Located to the right of the Environment Configuration File option)



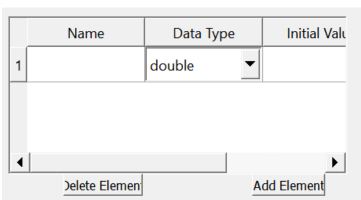
3. Type Environment Port (located to the right the Environment Configuration File Location option)



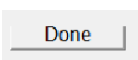
4. Determine the Environment Bounds (Located to below the Environment Port Option)



5. Add and delete elements as well as customizing the name, data type, and initial value (Located to the left of the Environment Bound option).

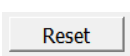


6. Select the Done Button when finished (Located at the bottom right)

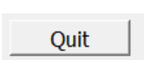


7. OPTIONAL: Select the Reset Button to reset all tabs (Located at the bottom to the right of the Done button)

Commented [MP3]: Having a step by step type list which tells you to select the reset button may not be a good idea.



8. Select the Quit Button to exit the VAST Configuration Wizard (Located at the bottom to the right of the window)



2.3 Configuration File Format

After running the initialization wizard, it will generate an XML file that looks like this.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<VAST xmlns:dt="urn:schemas-microsoft-com:datatypes">
    <module module="VAST">
        <map>
            <pair type="mandatory">
                <key>
                    <name>output_file_location</name>
                </key>
                <value>
                    <name>C:/Scripts/VASTConfig</name>
                </value>
            </pair>
            <pair type="mandatory">
                <key>
                    <name>viz_option</name>
                </key>
                <value>
                    <name>true</name>
                </value>
            </pair>
            <pair type="mandatory">
                <key>
                    <name>time_ratio</name>
                </key>
                <value>
                    <name>0.01</name>
                </value>
            </pair>
            <pair type="mandatory">
                <key>
                    <name>time_step</name>
                </key>
                <value>
                    <name>0.1</name>
```

This XML file is used by VAST to determine what the user wants in their Environment, how they want the AV positioned, where the AV is located, etc. This is what the XML file contains.

Table 1: Configuration File Elements

Name	**Module**	Description	Input
Output file location	VAST	Outputs the results of the runs(s)	File location
Visualization option	VAST	Option for VAST to run with a visualization	True or false
Time Ratio	VAST	Ratio of real time to accelerated time	Ratio

Time step	VAST	Value of the time step, aka how much time passes per step	Double value
Number of replications	VAST	Number of runs	Int value
Seeds	VAST	Seed values for the run	Int values separated by commas
Max run time	VAST	Time the repetitions run for	Double value
Metrics	VAST	The metrics that the user has selected	AvgSpeed, AvgAccel, AvgDecel
Environment Configuration File Location	Environment	The file location of the configuration file for the environment	Configuration file location
Environment Obstacle Port	Environment	Port that we use to communicate with the Environment	Port number
Exe location	Environment	Simulator exe location used for the Environment	File location
Environment Bounds	Environment	Bounds on the environment that the AVs cannot pass	x, y, and z values
AV Name	AV	Name of the AV	String

AV Movement Port	AV	Port that the AV uses to communicate with the Environment	Port number
Exe Location	AV	Script for the AV	File location
AV Location	AV	Initial location of the AV	x, y, and z values
AV Orientation	AV	AV Initial Orientation	Three degree values seperated by commas
AV Bounds	AV	AV Bounding Box (Used for collisions)	x, y, and z values
Sensors	AV	Default sensors + custom sensors the user adds by coding their own sensors	ProxSensor, CustomSensors...

In addition to the mandatory values listed above, there are some optional values that one can add an indefinite amount of. These option values have a name for the variable located between the <key> tags, followed by a <value> that contains the type of the value, plus the value itself.

```
<pair type="optional">
    <key>
        <name>test</name>
    </key>
    <value>
        <name>bool, True</name>
    </value>
</pair>
```

3 Important VAST Components

3.1 AV Example Class

The AV class is one of the parent classes used in . These fuctions allow AV to update itself to the Event Tree with any new information that the AV recieves about itself as well as gives the AV a general update function that can be used by any of the child classes that AV will have. The AV class itself provides the user with all the functionality they will need to create a link between the AV of their choice and the entire VAST program using a data map that can be accessed by any number of child classes.

These child classes will have to be implemented by the user themselves to include the type of AV that they are wanting to test. The overall class allows the user to implement anything that they want and have a way for their implementation to interact with the rest of the VAST program seamlessly.

```
#pragma once
#include "AV.h"
#include <iostream>
#include <fstream>

using namespace std;

class PythonVehicle : public AV
{
public:
	PythonVehicle(string vehicleID)
	{
		_vehicleID = vehicleID;
	}

	void retrieveInformation();

	void sendInformation();

	//calls this child classes functions
	dataMap callUpdateFunctions();


private:
	ofstream _output;
	ifstream _input;

	string _vehicleID;
	dataMap currentData;
};
```

This is the example of the child class used to run the python script.

3.2 Environment Example Class

The Environment is a child class of VComponent much like the AV which comes with all the functionality that the VComponent class has. The main usability that the Environment gains from the VComponent is the ability to use the same update function as all the other classes so that one function call can be used to call all the different classes updates. The parent class of Environment allows the user to have access to the data map that is associated with the environment as well as make changes to it.

To correctly use the Environment class, you will need to create a child of the class that will include all the functionality to the specific environment that you will be using. This means that while the original Environment class will be the same between every different simulation environment the user will have to declare the specific interaction with the simulation.

```
#pragma once
#include "TraCIAPI.h"
#include "Environment.h"

class SumoEnvironment : public Environment
{
public:
	SumoEnvironment(string fileLocation, int AVid, string SUMOexeLocation, int isRandom)
	{
		_fileLocation = fileLocation;
		random = isRandom;
		_AVid = AVid;
		_SUMOexeLocation = SUMOexeLocation;
	}

	//Opens the Sumo Environment with the file location
	void openEnvironment();

	//Gets the information from Sumo via traci commands
	void getMapInformation();

	//sends the new command to the AV in sumo if this is required
	void changeAVCommand();

	//calls this child classes functions
	dataMap callUpdateFunctions();

private:
	string _fileLocation;
	int random;
	string _SUMOexeLocation;
	string _AVid;
	TraCIAPI traci;
	dataMap currentData;
};
```

This is an example of the specific declaration of the Sumo Environment that is used in the demo.

3.3 Event Tree Class

The Event Tree class inherits from the VComponent class and like the classes mentioned earlier in previous sections. This gives the event tree class the ability to use the functionality that the VComponent class has. The functionality of the Event Tree is to collect and manage AV and environment events that occur in the simulation and constantly update the scenario clock.

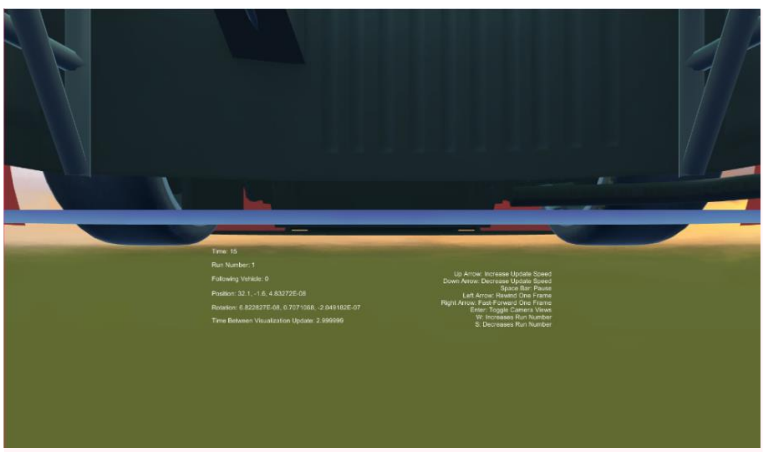
```cpp
using namespace std::this_thread;     // sleep_for, sleep_until
using std::chrono::milliseconds;
using std::chrono::microseconds;
typedef std::invalid_argument InvalidArgumentException;
typedef double ratio;
typedef map<VComponent*, dataMap> tableMap;

class EventTree
{
private:
	/* Exception for time clock discrepencies.*/
	class OutOfTimeException : public exception
	{
	public:
		/* Shows the error of adding an event out of sync with the sim clock.*/
		OutOfTimeException(timestamp currentTime, string offendingComponent, double offendingTime) :
			exception()
		{
			stringstream ss;
			ss << offendingComponent << " attempted to add an event at " << offendingTime
				<< " but the clock was at " << currentTime;
			_messageEnd = ss.str();
		}

		/* Shows the error of adding a VAST component when in the middle of replications.*/
		OutOfTimeException(string functionName, string offendingComponent, bool running)
		{
			stringstream ss;
```

```cpp
			if (running)
			{
				ss << "Replication Running";
			}
			else
			{
				ss << "No Replication Running";
			}
			_messageEnd = ss.str();
		}

		/* Overridden from parent std::exception.  Identifies this as an exception thrown
		by VAST for reasons specific to this system and the clock phases by which the system
		operates.*/
		virtual const char* what() const throw ()
		{
			stringstream ss;
			ss << "VAST EventTree cannot perform this action at this clock phase: "
				<< _messageEnd;
			return ss.str().c_str();
		}
	private:
		string _messageEnd;
	};

	/* Exception for database discrepencies*/
	class DatabaseException : public exception
	{
```

3.4 Unity Collision Detection

The collision detection is going be driven by Unity for the purpose of training and testing an AV. This will be done by detecting a collision between the AV avatar and the SUMO generated vehicles in the simulation. Unity will read the output generated by SUMO which contains vehicle information such as vehicle position, speed, etc. The number of vehicles in the simulation is determined by the SUMO output file. At the end of the simulation, the collision information is then outputted by Unity.

The user inputs key commands such as spacebar to pause, the up and down arrows to increase or decrease the update speed respectively. The left and right keys can be used to rewind and fast-forward the simulation respectively. The W and S keys can be used to increase and decrease the number of runs and enter is used to toggle the camera views.



3.5 Unity Post-Simulation Visualization

Commented [OCE4]: Need to be written

The Post-Simulation Visualization gives the user the ability to visualize a scene using Unity while the simulation is running. In the case of VAST, post simulation visualization is used to show a scene of the simulation and the AV. The image above shows the post simulation visualization being used to demonstrate the collision detection. The Post-Simulation Visualization contains simulation information, canvas with text rendering, and mesh rendering. Information of the simulation is retrieved from the output file from SUMO and a scene is created using that information. User input is used to update all user interface information which include the five-text object reference: time text, car following text, position text, rotation text, update time text. The image below shows the simulation information that is displayed in the Post Simulation Visualization.

References

[1] https://unity3d.com/unity?_ga=2.132657418.527601306.1556677500-361549076.1555482325

[2] https://sumo.dlr.de/userdoc/Sumo_at_a_Glance.html

URL to repository

https://github.com/SwiftSniper123/VAST-Test-Bed/tree/master/VAST