Ground Vehicle Reconnaissance Planning in VR-Forces

Brian Casey McCormick

Department of Modeling and Simulation, UAHuntsville

Mod 795 Final Report

Abstract

The objective for our project is to repeat the original project of Petty and Van Brackle [1]. The original project involved developing an algorithm that generates a reconnaissance route for a friendly ground vehicle to follow in order to sight hostile ground vehicles on a polygonal terrain database. A computer generated forces (CGF) tool was used to simulate the motion of the friendly ground vehicle following the generated reconnaissance route on the polygonal terrain. The CGF tool recorded various parameters such as the sighting time of the hostile ground vehicles to help evaluate the algorithm's effectiveness. Empirical and statistical analysis showed the algorithm performed reconnaissance route planning at the level of human experts.

Of course, just repeating the original project exactly would leave much to be desired. We also enhance the previous work by using a modern CGF tool called VR-Forces that supports modern terrain databases [3]. In addition, we improve the route planning algorithm to better handle terrain databases with few terrain features of interest. The statistical results indicate our algorithm is quite effective, sighting more than 99% of the hostile ground vehicles for two terrain databases. These results provide an extra validation to the results of Petty and Van Brackle project.

Introduction

We choose to use MAK's VR-Forces to repeat the Petty and Van Brackle project for a variety of reasons. First, VR-Forces is a modern CGF tool that provides the ability to simulate synthetic environments with urban, battlefield, maritime, and airspace activity. Next, VR-Forces supports the use of modern terrain databases. Also, VR-Forces has an easy to use interface for developing scenarios, running scenarios, and visualizing the simulation results. In addition, we can take advantage of its batched processing capabilities in order to run many simulations during our computer's idle time. Another very important reason we choose to use VR-Forces is its default functionality can be extended with Lua scripting [4].

As explained in Petty and Van Brackle, terrain reasoning is one of the most important features of a CGF tool such as VR-Forces. In this regard, VR-Forces performs well. It has several pre-defined tasks involving terrain reasoning that can be assigned to scenario entities such as ground vehicles. The names of these tasks include the following: 'Move to Waypoint (Direct)', 'Move to Location (Direct)', 'Move to Waypoint (On Roads)', 'Move to Location (On Roads)', and 'Move Along Route'.

We take advantage of the VR-Forces functionality called 'Move Along Route' for our project. This functionality can navigate our friendly ground vehicle along the route we specify in real time in accordance with the various obstacles in the terrain. It does so while considering the ground vehicle's dynamics and it performs basic collision avoidance with other entities. Figure 1 illustrates the usage of the 'Move Along Route' task. The diamond shapes represent our route points, the line represents the

path that VR-Forces might move the friendly ground vehicle, and the large blue circle represents an obstacle such as water.

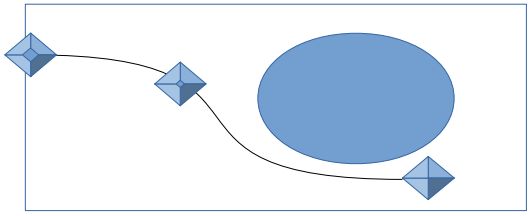


Figure 1. Notional VR-Forces Path Following

Use of VR-Forces

The manner in which we use VR-Forces is as follows. The interested reader can setup the same base scenario with assistance from the VR-Forces First Experience Guide and the VR-Forces User's Guide [5]. We first create a scenario and select a terrain database that comes with VR-Forces called 'mak-land'. The terrain database is interesting as it has both urban and rural features.

Next, we place a friendly ground vehicle of type M2A2 Bradley IFV on the terrain and assign a Lua script task to it. We will have much more to say about the contents of the Lua script task in this report. The Lua script task will begin running when the scenario starts running.

After this, we create a Route to indicate the map region of interest for the reconnaissance route. Note, this route needs to be named 'Route 1' to be consistent with the Lua script task we create. The Lua script task will automatically connect the start and stop points of the route when it uses it. Figure 2 shows a screen shot of our starting 'mak-land' scenario. The small blue icon represents our friendly vehicle and the red polygon represents 'Route 1'.

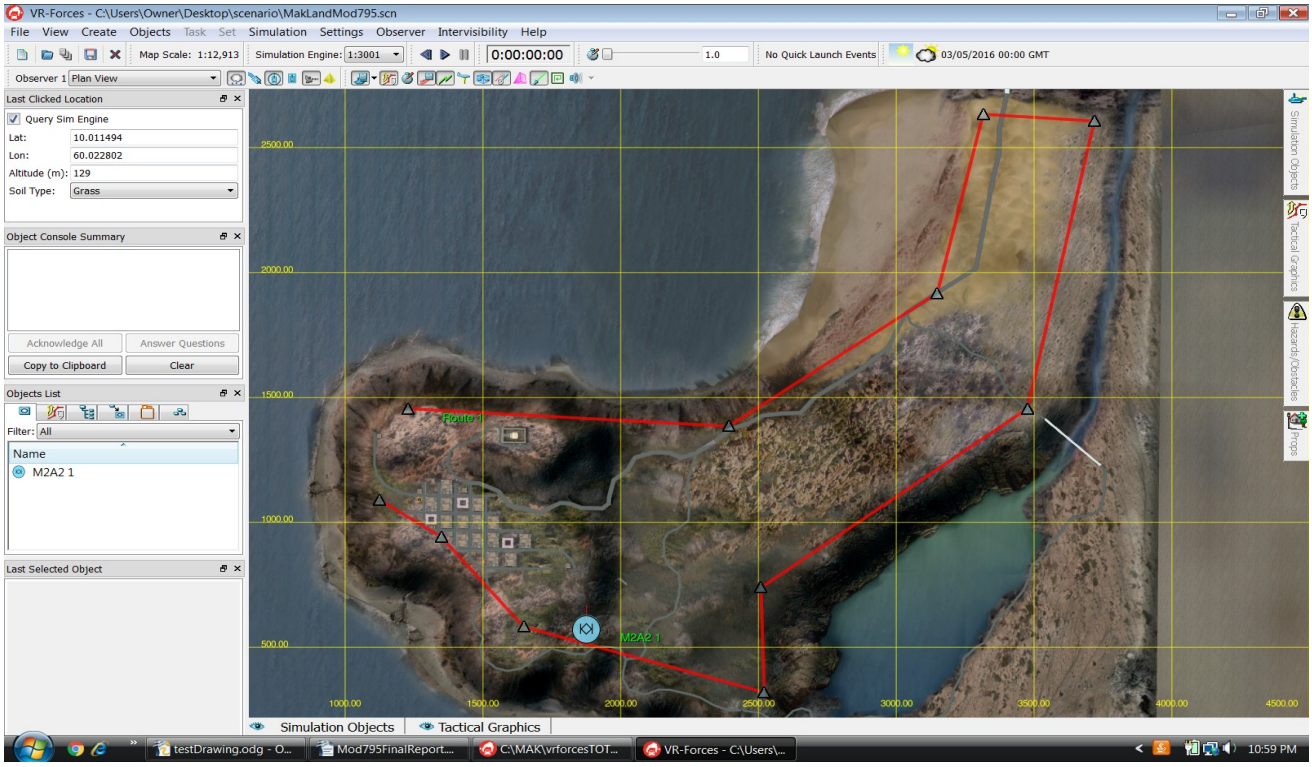


Figure 2

Figure 3 show a screen shot of our starting 'village' scenario. The same comments apply concerning the small blue icon and red polygon as in the 'mak-land' scenario. Though not easy to see on the top view, the 'village' terrain database is fairly mountainous and is therefore interesting for our project.

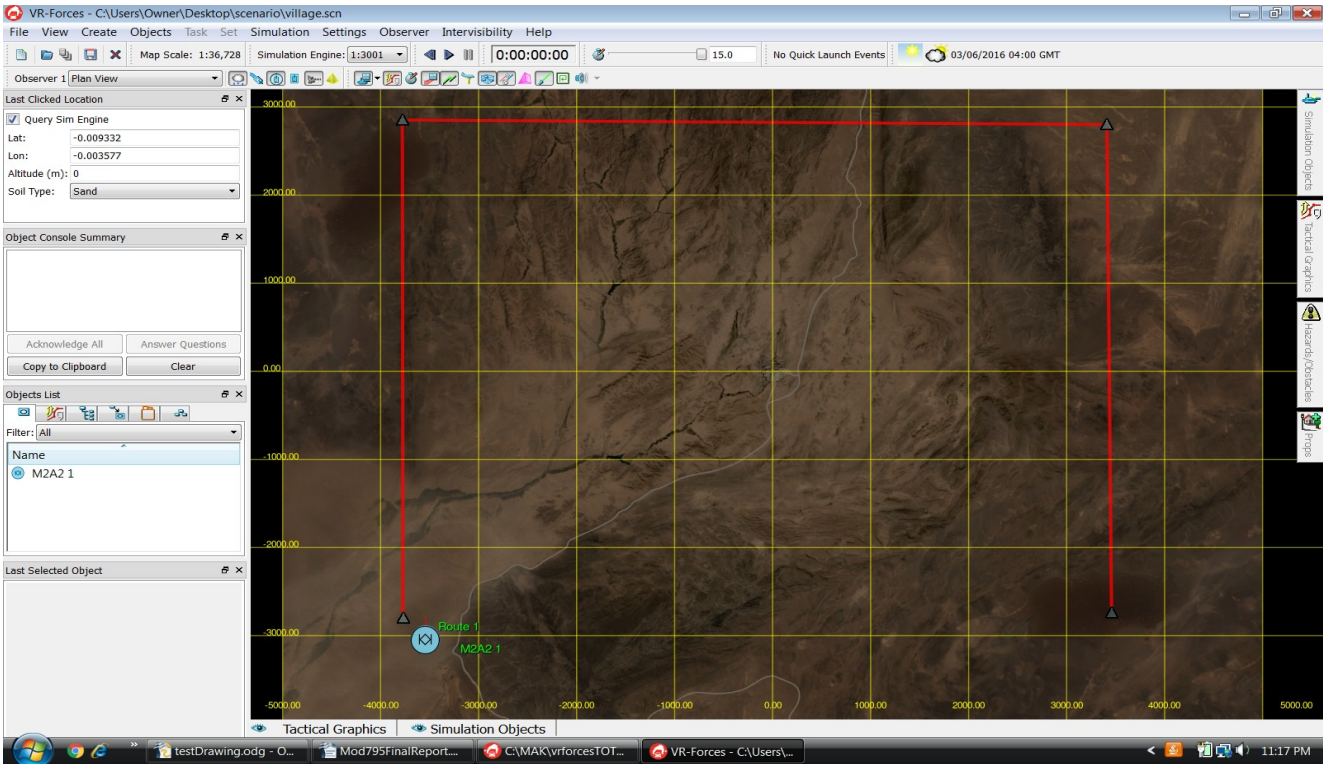


Figure 3

Now that we have our base scenarios, we use them for our simulation runs by taking advantage of VR-Forces batch file processing capabilities. We define a batch file as explained in the VR-Forces User's Guide. After the user enters the batch file for VR-Forces processing, VR-Forces proceeds by closing any open scenario, loading the scenario we wish to run, running the scenario for our desired run duration, and stopping the scenario. The entire process is repeated for our desired number of simulation runs. We will use batch file processing to perform our fairly lengthy simulation runs during our computer's idle time.

The Lua script task is the real workhorse for a scenario run. The Lua script task places 20 hostile ground vehicles of type GAZ-69 Utility Vehicle at random locations on the terrain. Also, it queries the terrain database for various terrain features such as buildings and trees. It then uses these features to generate potential route points in the terrain, along with determining their visibility with respect to each other. Next it generates a reconnaissance route according to the algorithm from Petty and Van Brackle for the friendly ground vehicle to follow. The Lua script task then commands VR-Forces to simulate the friendly ground vehicle's movement along the route. Last, it records the hostile ground vehicle sightings to a file. The file name corresponds to the system clock time at the start of the scenario run.

VR-Forces Simulation Items

We now highlight the simulation items we created for this project. We will provide our simulation items to the interested reader upon request [8]. The description is intended to give a bird's eye view of everything that is needed for our simulation runs. The interested reader can duplicate our simulation

runs by replacing their Lua script task contents with the contents of Mod-795-Alg-Waypoints.lua and adding the remaining Lua files that we highlight here to their local Mak 'script' directory. Our directory is located at C:/Mak/vrforcesTOT20160111/userData/scripts. We also provide a sample batch file called sampleScenarioBatch.bsn.

(1) Mod-795-Alg-Waypoints.lua – this is the Lua script task that loads the map region of interest, queries for the terrain database features, generates the vertex and edge information for the reconnaissance algorithm, calls the function ComputeOrderedVertices(vertices, initialLat, initialLon, fileIn) in graph.lua to generate the reconnaissance route, creates and adds the route to the scenario, adds 20 randomly placed hostile ground vehicles on the map region of interest, moves the initial friendly ground vehicle to a random location, commands the friendly ground vehicle to follow the route, and records the hostile ground vehicle sightings to a file

(2) pointInPoly.lua – contains a Lua support function that determines if a 2d point is contained in a 2d polygon, the polygon can be either concave or convex, the VR-Forces Lua API does contain a function for determining if a point is inside a polygon, but it only works for concave polygons

(3) graph.lua – contains the Lua function ComputeOrderedVertices(...) and its supporting Vertex and Edge functions which act very much like object-oriented classes, it implements the reconnaissance algorithm described in this report

(4) split.lua – we extracted this Lua support function from the web, as indicated by its name, it provides functionality to split a string into substrings in accordance with an input delimeter [6]

(5) struct.lua – we extracted this Lua support function from the web, as indicated in its name, it provides a clean way to use structs in Lua that closely resemble C style structs, without this function the handling of structs in Lua is much more cumbersome [7]

(6) deterministicEdges.lua – this contains a Lua support function that reads vertex-to-vertex visibility information from a file and another Lua support function that outputs the vertex-to-vertex visibility for two input vertex id's

(7) CountSightings.java – this is a Java source file to read and parse the data files generated for a single simulation run, it counts and displays the number of hostile ground vehicle sightings

(8) sampleScenarioBatch.bsn – a sample batch file that VR-Forces can load

Note, we do not provide the actual code changes that we made to Mod-795-Alg-Waypoints.lua for our project. We had to make a small change to remove the slight variations in the results of the call to the VR-Forces Lua API doesChordHitTerrain(Location3D, Location3D) function. Oddly enough, sometimes the results for this function are not deterministic for the same inputs. We discuss the details of the small change we made in the 'Results' section of this report. We assume the reader is not worried about these slight variations however and prefers a Lua task script that can be run 'out of the box'.

Reconnaissance Route Planning Algorithm

The first algorithm we discuss analyzes the map region of interest and selects a set of route points that should be visited. The second algorithm determines the order in which the set of route points should be visited. The algorithms are based on the All-Points algorithm in the Petty and Van Brackle project. Our algorithm contains a slight modification to the route point selection algorithm to better handle

cases in which there are few terrain features of interest.

Route Point Selection Algorithm

The goal of the route point algorithm is to analyze an input map region of interest and determine its route points. In order to do so, it needs to consider the set of so-called Important Points in the All-Points algorithm. Important Points are ground points of significant interest in the terrain, such as urban dwellings or trees. These points are important for reconnaissance as they are typically the most difficult points to see in the terrain.

The algorithm includes ground points that were not defined in the Petty and Van Brackle project that we call Grid Points. These points are obtained by performing a rectangular gridding of the map region of interest. These points are important. First, they allow more ground points to be considered for the final route point selection and hence allow for potentially better routes to be generated. Second, they may be the only ground points that get considered for the route if there are few or no terrain features of interest in the terrain database. Note, the algorithm does check to make sure a gridded point can be navigated to by the friendly ground vehicle before allowing it can become a Grid Point.

While the Grid Points are generated just by evenly gridding the map region, the selection of the Important Points requires more work. The algorithm loads terrain features with the VR-Forces Lua API for the map region of interest. It then culls out features that are not expected to cause visibility issues such as water, while retaining features that would likely cause visibility issues such as urban dwellings or trees. The algorithm creates ground points very near the Grid Point that would likely cause visibility issue. The set of these ground points are called the Important Points.

Next, the algorithm creates a vertex for each Grid Point and Important Point. The vertex contains the latitude/longitude from the associated point and is assumed to be located on the terrain surface. The vertices originating from a Grid Point simply uses the latitude/longitude values directly, but the vertices originating from an Important Point cannot do this. Instead, the algorithm creates 2 vertices a short distance north and south of the Important Point, ensuring the selected points can be navigated to by the friendly ground vehicle before allowing them to be a Grid Point. We learned VR-Forces does not make a 'best effort' and get as close as it can to an unnavigable route point. Instead it just skips navigating to the route point entirely. The '2 vertices' change overcomes this issue.

Now we have a collection of vertices on the map region of interest and we need to decide which to select as route points. The algorithm attempts to select the minimum subset of vertices that have visibility to each other. This is the NP-Hard minimum dominating set problem, so we wisely use a greedy algorithm strategy that runs quickly and gives sufficient, but somewhat sub-optimal results.

The selection of the vertices that will become route points is accomplished by first constructing a graph where the vertices are as previously described and edges are added between any 2 vertices if they have line of sight to each other. The VR-Forces Lua API contains a function to determine if one location can see another location, called doesChordHitTerrain(...). We apply this function to every pair of vertices and add an edge to the graph representation when they have visibility between them. The algorithm assumes the starting vertex is at a height of 2.5 meters above the terrain and the stopping vertex is at 1.0 meters above the terrain, the same as in the Petty and Van Brackle project. The resulting graph is called a line of sight graph.

Note that when assigning visibility to the vertex pairs that each vertex that originates from an Important Point has visibility to itself and thus has an edge originating from and pointing back to itself. The

vertices originating from the Grid Points do not have an edge pointing back to themselves. This may seem strange, but it is necessary to ensure all the Important Points are either picked to be a route point themselves or are visible from a vertex that is picked to be a route point. There is no real need to make sure each Grid Point is either picked as a route point or visible from another route point.

The algorithm continues by determining which vertex has the best visibility, that is, the vertex that has the most edges. This vertex is added to the route point list. After this, the algorithm removes all the edges pointing at this vertex. It then also removes all the edges pointing to all of this vertex's neighbors. It does so because visibility to the neighbor vertices is no longer necessary. This vertex has become a route point and can see it. The algorithm continues by repeating the selection of the next vertex to be a route point in the same manner until there are no more edges.

Route Point Ordering Algorithm

At this point, we have a set of route points that must be visited. The goal of this algorithm is to determine the optimal order to visit the route points. We use the exact algorithm logic as done in the Petty and Van Brackle project, who based their algorithm on an earlier paper by Rosenkrantz and Stearns [2]. The main idea behind the algorithm is the shorter the resulting route path length is, the better it is for reconnaissance. Hence the algorithm tries to obtain a path that minimizes the path length. We now describe this algorithm.

We first point out that we have a small deviation in the Petty and Brackle version of this algorithm. We don't set the initial route point to the northwest corner of the map region, but instead we set the starting vertex to the initial location of the friendly ground vehicle. The algorithm adds this route point to the selected route point list. The algorithm then picks successive route points to move from the unselected route point list to the selected route point list by determining which unselected route point has the largest average distance to the already selected route points.

It is easy to see that the second route point that the algorithm will select will be the route point furthest from the friendly ground vehicle's initial position. Handling of the edges for this is a special case as it automatically adds edges between them in order to complete the initial Hamiltonian circuit. A notional example of this is shown in Figure 4. Note FV/RP1 stands for friendly ground vehicle, RP1 stands for route point1, and RP2 stands for RP2.

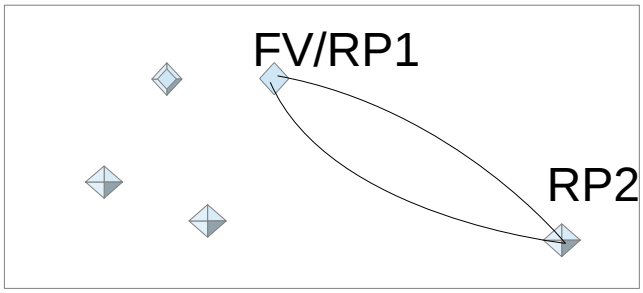
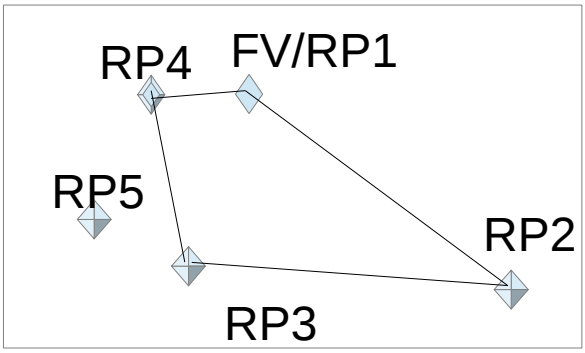
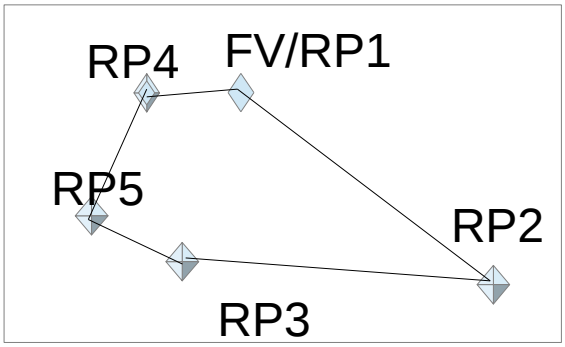


Figure 4

From here, the algorithm continues selecting route points as indicated above until all the unselected route points have been moved to the selected route point list. The addition of each route point into the Hamiltonian circuit is made so that the Hamiltonian circuit remains a Hamiltonian circuit. It does this by determining between what two selected route points could the route point under consideration be inserted that minimizes the increase in the path length. It then deletes the existing edge between those nodes and adds new edges from the selected route points, each pointing to the newly added route point.

Figures 5 and 6 shows a notional example of the algorithm choosing RP3 and RP4 to insert RP5 between. It is clear that trying to insert RP5 between any other selected route points would not only increase the path length unnecessarily, but it would also create a very awkward path to follow. Note when RP5 is added to the Hamiltonian circuit, the edge between RP3 and RP4 was removed and new edges were added between RP4/RP5 and RP3/RP5, keeping the Hamiltonian circuit complete.





Figures 5, 6

This algorithm is greedy and does produces sub-optimal results. However, since generation of the optimal path is equivalent to solving the traveling salesperson problem, we happily trade-off sub-optimal, fast results for optimal, slow results.

Results

We need to first mention that we had to modify Mod-795-Alg-Waypoints.lua at this point in order to deal with the slight variations issue discussed in the 'VR-Forces Simulation Items' section. First, we made a single simulation run to output the vertex-to-vertex visibility to a file. We did this immediately after the call to the doesChordHitTerrain(...) function. We output the visibility information in the following form:

```
--the following sequence repeats for all vertex-to-vertex combinations
vertexIdStart_vertexIdStop [true/false]
```

Next, we added a call to the ReadEdgeInfo() function in the init() function in order to load the vertex-to-vertex visibility at the start of the simulation run. After this, we replaced the call to the doesChordHitTerrain(...) function with a call to the EdgeIsVisible(vertexIdStart, vertexIdStop) function to query the pre-determined visibility data. Both the ReadEdgeInfo() function and the EdgeIsVisible(...) function are located in deterministicEdges.lua. Making this change ensured that our vertex-to-vertex visibility information was deterministic and consequently massively reduced our required run matrix.

After completing our adjustment to Mod-795-Alg-Waypoints.lua to ensure vertex-to-vertex visibility remains constant from run to run, we are assured that any initial set of ground vehicle locations will give deterministic simulation results in VR-Forces. Given this, we use the popular Monte Carlo simulation technique for our batched simulation runs. In particular, we generate random locations for the friendly and hostile ground vehicles for each simulation run for our 'mak-land' and 'village' scenarios. We then statistically analyze the number of sightings over all the runs for both scenarios to help evaluate the effectiveness of our algorithm.

The results of 30 'mak-land' terrain database simulation runs show the generated reconnaissance routes are extremely effective. We had 28 of the 30 runs with 20 out of 20 hostile ground vehicle sightings and 2 of the 30 runs had 19 out of 20 hostile ground vehicle sightings. This gives an average of 19.93 hostile ground vehicle sightings with a standard deviation of 0.25 hostile ground vehicles. An example of the resulting route for this terrain is shown in Figure 7.

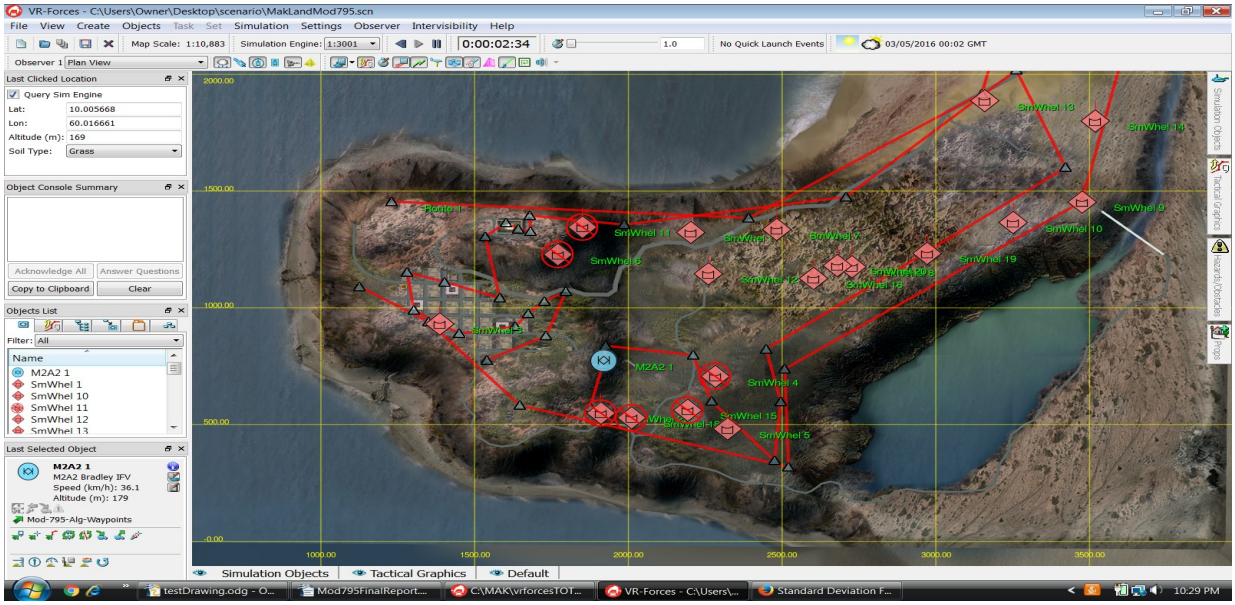


Figure 7

The results of 30 'village' terrain database simulation runs show quite similarly impressive results. We had 29 of the 30 runs with 20 out of 20 hostile ground vehicle sightings and 1 out of 20 runs had 19 out of 20 hostile ground vehicle sightings. This gives an average of 19.97 hostile ground vehicle sightings with a standard deviation of 0.18 hostile ground vehicles. An example of the resulting route for this terrain is shown in Figure 8.

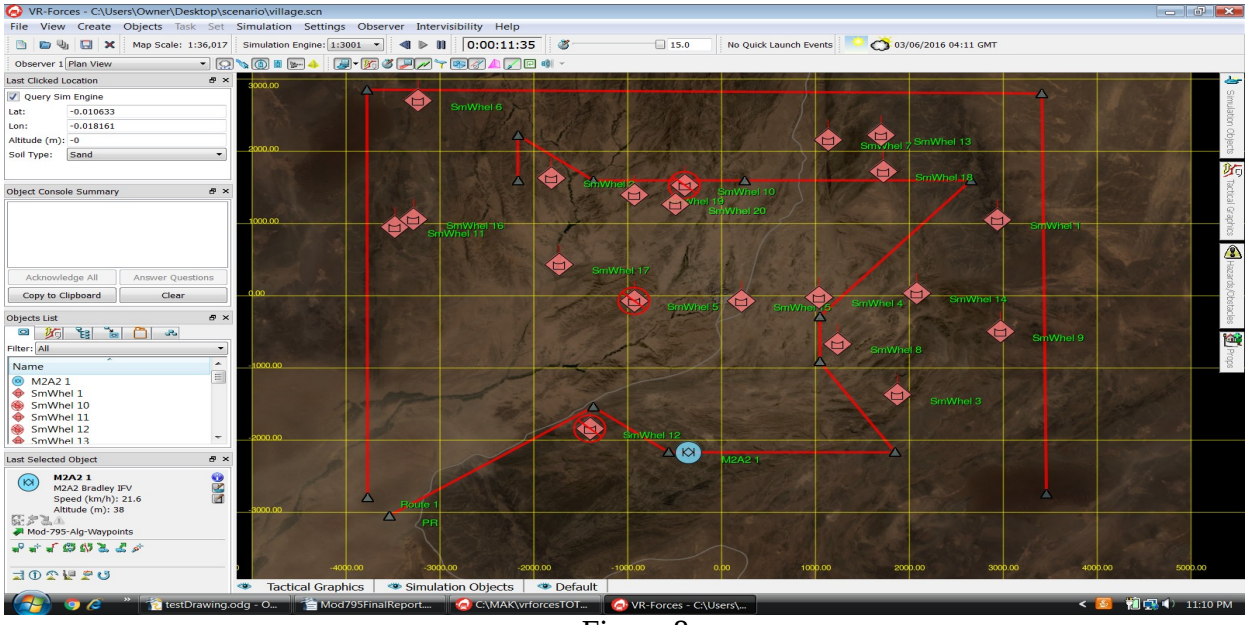


Figure 8

We investigated the cause of the missed hostile ground vehicle sightings and highly suspect it is due to the occasional problem that a route point was not visited by the friendly ground vehicle. We suspect this is caused by the friendly ground vehicle not being able to physically reach a given route point or the route point happens to be inside a region that VR-Forces considers to be an obstacle.

Moments of Frustration Using VR-Forces

While VR-Forces is a very good CGF tool that met the needs of our project well, we did experience several moments of frustration. We list our moments of frustration, in no particular order.

(1) We had a difficult time making the vertex-to-vertex visibility the same from run to run. We even tried adding a 30 second delay after the feature data is available to allow the system to 'settle' its map data loading, to no avail. It would be nice if the results of the VR-Forces Lua API function doesChordIntersectTerrain(...) were deterministic.

(2) We had a difficult time defining a friendly ground vehicle route in the Lua script due to the interface for creating a Location3D requiring the altitude to be in MSL. It would be very convenient if there were a way to create a Location3D with the altitude in AGL.

(3) We had a difficult time testing new Lua scripts for correctness when they are running as part of VR-Forces Lua script tasks. This issue forced us to download and compile Lua ourselves in order to create a stand-alone Lua run/test environment. It would be nice if VR-Forces provided the ability to run/test Lua scripts in a stand-alone fashion.

(4) We had a difficult time creating the hostile ground vehicles from within the Lua script task. We had to look at the property window for the hostile ground vehicle we wanted to create in VR-Forces to determine the appropriate 'entity_type' attributes we needed to create it. However, even then we still had trouble as the 'force' parameter value required quotes around it. We resolved this issue by trial and error. There isn't a single example of setting the 'force' parameter in the VR-Forces Lua API documentation. In general, it would be helpful if the documentation provided an exhaustive collection of examples for performing such tasks.

(5) We had a difficult time understanding the simulation results due to the console panel only displaying 100 lines of output at a time. It was only when we redirected the output to a file did we realize what was happening. It would be nice if VR-Forces allowed the user to display more than 100 lines on the console panel.

(6) We had a difficult time figuring out why our console panel 'info' messages were not actually being output to the console panel. It turned out the console was only displaying 'warning' messages. We eventually figured out how to fix this by selecting the friendly ground vehicle and using the pulldown menu on its information dialog box. It would be nice if VR-Forces had the same pulldown menu on the console panel itself.

(7) We had a difficult time with the use of the VR-Logger tool. We initially wanted to use the tool to record all data going to the console panel. However, when we ran our scenario with the VR-Logger running, nothing was ever recorded. We looked at the VR-Forces User's Guide documentation for assistance, but it did not provide enough information to understand how to use it. We eventually concluded the logger was probably just recording DIS message traffic and not the console output, so we stopped using it and started logging the simulation run results to a file instead.

(8) We had a difficult time figuring out how to use Lua script tasking in VR-Forces for the situation in which the primary Lua script task calls our other Lua script functions. We eventually figured out our other Lua scripts had to be placed in the Mak/vrforcesTOT20160111/userData/scripts directory. In general, it would be nice if the VR-Forces documentation provided more details on the use of Lua script tasks for more extensive projects such as ours.

Future Work

We now list the ways our desires for extending our work below, in no particular order.

(1) We would like to resolve the issue in which the friendly ground vehicle does not visit a route point.

(2) We would like to allow more than 1 map region of interest to be defined and have the algorithm generate a reconnaissance route for the entire set of them.

(3) We would like to have multiple friendly ground vehicles defined in the scenario and generate an optimal route for each friendly ground vehicle that maximizes the overall effectiveness.

(4) We would like to compare our generated reconnaissance routes to those generated by human experts as was done in the Petty and Van Brackle project.

(5) We would like to record more data and use it to perform a more extensive statistical analysis of the results as was done in the Petty and Van Brackle project.

(6) We would like to create a more robust way to define the route on the map region of interest. Specifically, the route that is generated should not have to be named 'Route 1' to work correctly with our Lua script task. This could be done by creating a user interface component for selecting the route name during the creation of our Lua script task.

(7) We would like to evaluate our algorithm against the same terrain databases with very few hostile ground vehicles that are placed in locations that have limited visibility.

Conclusions

We successfully implemented a reconnaissance route generation algorithm based on the Petty and Van Brackle project using VR-Forces. We extended the default functionality of VR-Forces using Lua scripting and used existing VR-Forces terrain databases for our scenarios. The reconnaissance algorithm was shown to be very effective as the friendly ground vehicle successfully sighted over 99% of the hostile ground vehicles in both the 'mak-land' and 'village' terrain databases over a statistically significant number of runs.

Acknowledgment

We want to extend a sincere 'thank you' to MAK representatives for providing a short-term academic license for using VR-Forces for this project.

References

[1] Petty MD, Van Brackle DR. Reconnaissance route planning in polygonal terrain written. In Proceedings of the 5th International Training Equipment Conference. The Hague, The Netherlands,

April 26-28, 1994, pp. 314-327.
[2] Rosenkrantz, D. J., Stearns, R. E., and Lewis, P. M. (1974). “Approximate Algorithms for the Traveling Salesman Problem”, Proceedings of the 15th Annual Symposium on Switching and Automata Theory, New Orleans LA, 1974, pp. 34-42.
[3] http://www.mak.com/
[4] http://www.lua.org/
[5] http://www.mak.com/support/resources/product-user-guides
[6] http://lua-users.org/wiki/SplitJoin
[7] http://lua-users.org/wiki/StrictStructs
[8] bcm0001@uah.edu; pettym@uah.edu