



Australian Government
Department of Defence
Defence Science and
Technology Organisation

Design of a Foliage Penetrating LADAR Simulation Tool

Mark Graham

Intelligence, Surveillance and Reconnaissance Division
Defence Science and Technology Organisation

DSTO-GD-0577

ABSTRACT

A simulation tool was developed using MATLAB and its Graphical User Interface Development Environment (GUIDE) to simulate aspects of an airborne foliage-penetrating Laser Detection and Ranging (LADAR) system in scenarios designed to contribute towards the military operational use of such a system. In particular, the simulation tool is intended for conducting analysis on how best to task the aircraft and position the sensor. This document provides an overview of the graphical user interface and software including: the design challenges; dealing with the input scenery data; modelling the sensor and platform flight path; and planned analysis of the simulation results.

RELEASE LIMITATION

Approved for public release

Published by

*Intelligence, Surveillance and Reconnaissance Division
DSTO Defence Science and Technology Organisation
PO Box 1500
Edinburgh South Australia 5111 Australia*

Telephone: (08) 8259 5555

Fax: (08) 8259 6567

© Commonwealth of Australia 2009

AR-014-521

May 2009

APPROVED FOR PUBLIC RELEASE

Design of a Foliage Penetrating LADAR Simulation Tool

Executive Summary

This document outlines the development of software, designed using the MATLAB programming language, which aims to accurately simulate aspects of an airborne foliage-penetrating LADAR system. This work was undertaken as part of Task 07/105 *Support to Intelligence, Surveillance and Reconnaissance* to investigate the tasking of airborne sensors to detect inconspicuous targets, particularly those partially obscured by forest canopies.

Laser Detection and Ranging (LADAR) three-dimensional (3D) imaging systems could be extremely useful for this application. Such systems form 3D images based on time-of-flight of laser photons, some of which pass through gaps in foliage or other partial obscurants such as camouflage nets. Hence the 3D image will contain partial information about any objects behind or underneath such obscurants. The obscurant can be removed from the image by applying a range threshold, leaving a partial image of the hidden objects which may include targets of interest. An improved overall image can be formed by combining images taken from several different viewpoints, using knowledge of the LADAR sensor's location at each viewpoint.

The aim of the work under this Task is to recommend how to task an aircraft such as a Tactical UAV with a LADAR imaging payload, recognising that such a system is currently hypothetical and has not entered the Defence Capability Plan. There was no intention to produce a detailed model to aid in designing LADAR systems for aircraft or for testing data processing algorithms; rather, the fidelity of the model was only intended to be suitable for operations research studies.

The scope of this document is the design of software and algorithms to run simulations of LADAR imagery acquisition. A graphical user interface has been developed for the simulation tool to make it easy to set up scenarios, modify parameters, visualise the output and perform analysis. An overview is also presented on the three-dimensional scene and target models used by the simulation and the modelling of the sensor and aircraft flight path. There were also several design challenges specific to the MATLAB environment which is discussed in this document.

Contents

| | |
|--|-----------|
| 1. INTRODUCTION..... | 1 |
| 2. ANALYSING SIMULATION RESULTS..... | 2 |
| 2.1 Potential Scenarios | 2 |
| 2.2 Measures of Effectiveness..... | 2 |
| 3. GRAPHICAL USER INTERFACE DESIGN OVERVIEW | 3 |
| 3.1 Layout and Design..... | 3 |
| 3.2 Program Control and Status..... | 4 |
| 3.3 Sharing Program Data using the Handles Structure..... | 5 |
| 3.4 Displaying Data | 6 |
| 3.4.1 Plotting the Scene | 6 |
| 3.4.2 Previewing the Flight Path..... | 7 |
| 3.5 Output Visualisation..... | 8 |
| 3.5.1 Limitations with Three-Dimensional Plots..... | 8 |
| 3.5.2 Applying a Threshold to the Output..... | 9 |
| 4. PROGRAM OVERVIEW..... | 10 |
| 4.1 File Setup..... | 11 |
| 4.2 Defining Parameters..... | 12 |
| 4.3 Running and Saving the Simulation..... | 12 |
| 4.4 Comparing the Output..... | 13 |
| 5. SCENE MODEL DATA | 14 |
| 6. FLIGHT PATH MODELLING | 15 |
| 7. SENSOR MODELLING..... | 16 |
| 7.1 Defining the Sensor..... | 17 |
| 7.2 Setting-up the Ray Data..... | 17 |
| 7.3 Preparation for Ray Tracing..... | 19 |
| 7.3.1 Defining the Search Space | 19 |
| 7.3.2 Restricting by z Value: Layers | 19 |
| 7.3.3 Restricting by x-y Values: Bounding Boxes | 20 |
| 7.3.4 Multiple Bounding Boxes..... | 22 |
| 7.4 Ray Tracing Algorithm | 25 |
| 8. ACKNOWLEDGEMENTS | 27 |
| 9. REFERENCES | 27 |

1. Introduction

Laser Detection and Ranging (LADAR) systems are used to produce three-dimensional images of the real world. LADAR sensors work by measuring the time it takes for transmitted laser pulses to return to a detector. A typical sensor will consist of multiple detectors arranged in an array. Each detector has its own timing circuitry which is used to measure the arrival times of photons returned from flood-illuminating the scene with a laser pulse [1]. This information is used to determine the range and consequently the spatial location in three-dimensional space of points that form a three-dimensional image of the scene.

This type of system can be utilised in a military surveillance context to detect and identify targets that are partially obscured, e.g. by foliage or camouflage netting. Returns are received from partially obscured objects because some transmitted laser photons will pass through the gaps in foliage or other obscurants and return to the detectors. It is possible to restrict a received image to a desired range of depths and so reveal any objects beneath or behind these partial obscurants. However, it is necessary for a foliage penetrating LADAR system to take and combine multiple images from different views through the foliage in order to form an overall image that will provide sufficient resolution for recognising objects. Different views can be collected by fixing the sensor in a single position and taking multiple images of a scene over time as the wind causes movement in the foliage [2], or by mounting the sensor to a moving platform such as an aircraft [3].

The software described in this document is intended to simulate a LADAR system on an aircraft. It uses three-dimensional point-cloud models to represent the real world as a set of voxels (volume elements in three-dimensional space). These scene models consist of the visible features of an area of land, including natural elements such as landforms and trees, and human elements such as vehicles and buildings. LADAR images are modelled by capturing the set of voxels that are within the sensor's field of view and have a direct line-of-sight to the sensor. Images taken from different viewpoints are then combined into an overall image. The purpose of the software is to analyse a variety of different flight paths and determine how best to fly the aircraft and position the sensor in order to detect and identify targets such as vehicles.

2. Analysing Simulation Results

The main purpose for developing this software is to assist in making recommendations for the operational use of an airborne foliage penetrating LADAR system. This will involve generating a set of realistic scenarios and comparison measures.

2.1 Potential Scenarios

Comparisons will be made by running a variety of scenarios which may include: circling around a possible target (Figure 1a); flying past a possible target (Figure 1b); and varying the standoff range (Figure 1c). Note that in these particular scenarios, it is assumed that the UAV has been cued to the approximate location of the target.

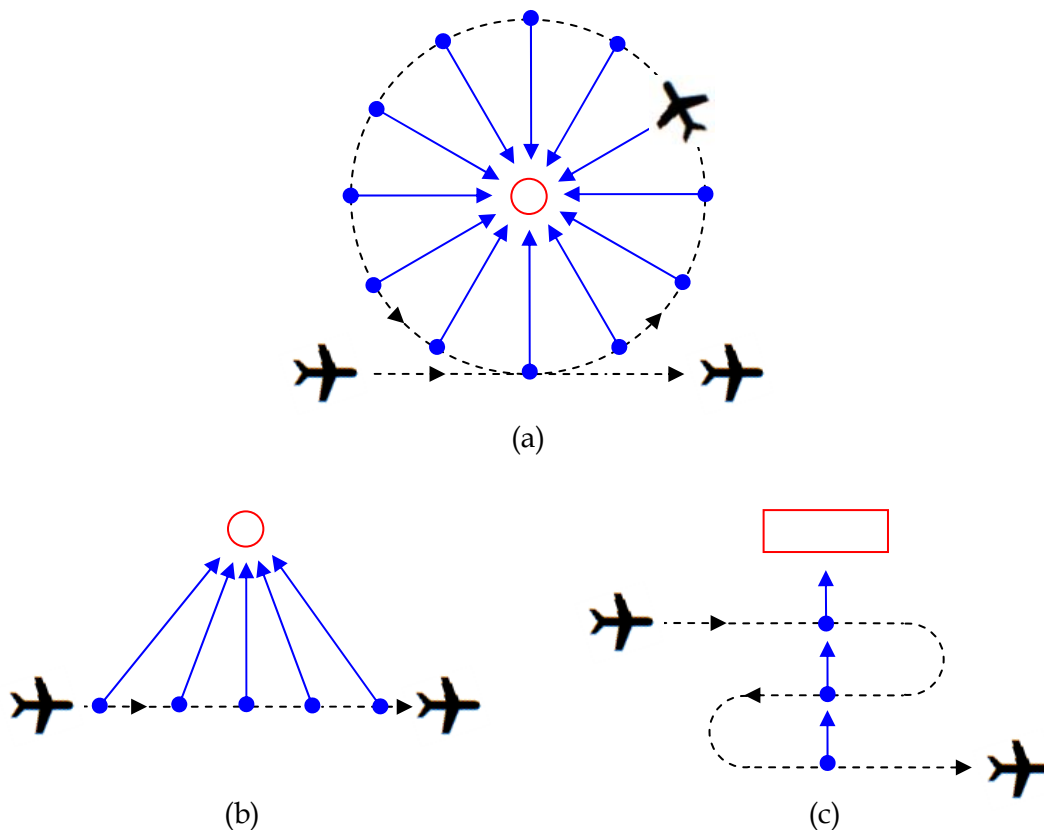


Figure 1: Top view of some flight paths to be considered. The targeted area is shown in red and the blue arrows indicate individual scanning positions.

2.2 Measures of Effectiveness

One mechanism for comparing scenario output results is visual, i.e. stating that one output of a target “looks better” than another. However this is a very subjective measure and to be able to make a meaningful comparison between different scenarios it will be necessary to devise some quantitative measures of effectiveness.

Possible measures of effectiveness could describe the total target coverage, distribution of target coverage or difference from an ideal result. For many flight paths the ideal result is not the complete target model but instead the view of the target that would be obtained if there were no foliage present. Let us call this view of the target the *unobscured view* and call the result with foliage present the *obscured view*. Assuming that an unobscured view of the target would be sufficient, a suitable measure of effectiveness might describe the difference between the unobscured and obscured views. The lowest expected difference would correspond to the most effective flight path. To support such measures of effectiveness, three-dimensional images of both the obscured and unobscured views of the target are generated.

Other measures that could also be investigated include the target coverage required for identification and the cost (in terms of time and distance) of flying different paths.

3. Graphical User Interface Design Overview

A graphical user interface (GUI) was developed for the simulation tool so that it would be easier to configure and run simulations as well as visualise the output and perform analysis. The software was written in MATLAB¹ and the GUI was designed using the Graphical User Interface Development Environment (GUIDE) which provides a set of tools to simplify the layout and programming processes of GUI design in the MATLAB environment.

3.1 Layout and Design

The layout and design of the GUI is simple and intuitive. Figure 2 shows the main window of the GUI which was constructed using standard components in the GUIDE Layout Editor.

The main functions of the program, such as loading files, are accessed through the menus and toolbar buttons across the top of the window. Simulation parameters, status indication and screen layout options are placed in frames along the left side and across the bottom of the window. This allows the user to view and make on-the-fly changes to these parameters and settings when appropriate. Additional popup windows are used for some operations, such as defining the input/output files (Figure 9) and displaying comparison results (Figure 10), in order to reduce clutter in the main window.

¹ The MathWorks, Inc., Natick MA, USA

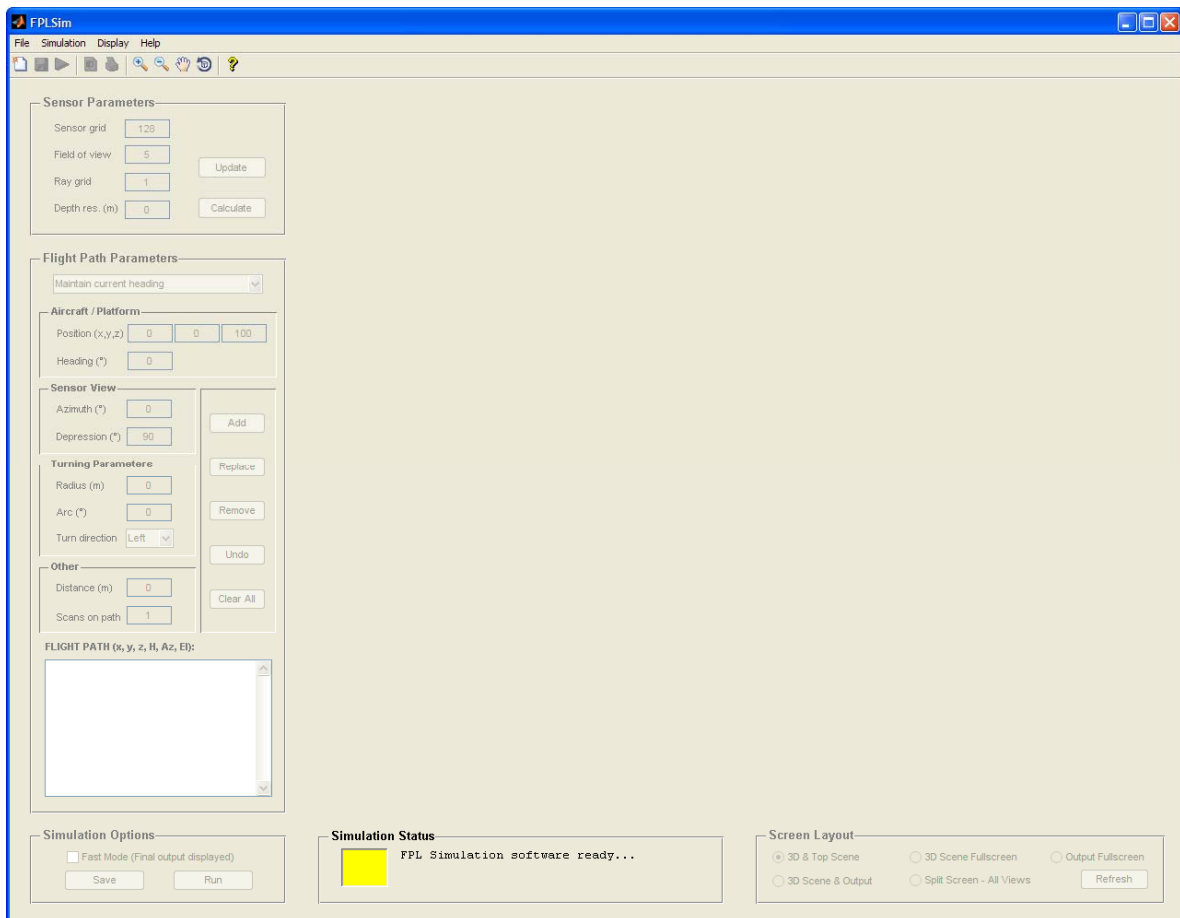


Figure 2: The main window of the LADAR simulation tool GUI. Graphical output such as the input scene model and output is displayed in the blank area of the window.

3.2 Program Control and Status

The development of the GUI focused on the areas of simplicity, functionality and user friendliness. To achieve these objectives required careful design to ensure a smooth flow of the program through all stages from setting up a scenario to running, generating and finally analysing the output.

Controls in the main window are greyed out and inactive when they are not available. This helps ensure that operations are performed in sequence and that it is not possible to perform illegal operations which may cause erroneous results. Most parameters for defining the sensor and flight path are set by entering values in textboxes. These values are checked to ensure they are numeric and within practical limits. If an invalid value is entered, the parameter is reset to its default value and an error message box may be displayed.

Visual status information is included in the software because of the lack of any suitable indication that a MATLAB program or block of code is executing (e.g. the mouse pointer does not change to the busy state). This information is located in the centre panel along the

bottom of the main window. A coloured box and text message are used to provide the user with information relating to the current program status. There are three unique program states as outlined in Table 1.

Table 1: Visual status indication of the program

| Visual status indicator | Current program status |
|-------------------------|--|
| Yellow | A yellow indicator is displayed when the program is idle but not all of the setup requirements have been met. This may include data files not loaded or simulation parameters not defined, which will be indicated by the text message displayed. |
| Green | A green indicator is displayed when all setup requirements have been met and the simulation is ready to run. |
| Red | A red indicator is displayed when the program is busy, such as when a data file is being loaded or the simulation is running. A percentage progress will also be shown when applicable. While the indicator is red the user is unable to do anything within the program. |

3.3 Sharing Program Data using the Handles Structure

Using GUIDE to design and create a GUI simplifies the programming process since a partially-complete file of MATLAB code (m-file) is automatically generated to control how the GUI operates. This includes code to initialise the GUI and a framework for *callback functions*, i.e. routines that are executed when the user interacts with components of the GUI.

All graphics object properties, including those used for implement plotting and visualisation functions, are accessed via *object handles* which serve as unique identifiers for graphics objects. GUIDE uses a variable called the *handles structure* to store the handles of all components in the GUI; a *structure* is a means of grouping and storing any type of data in named fields of a single variable. The handles structure is automatically passed to every callback as an input and a function `guidata` is used to maintain this structure (Figure 3). An unfortunate consequence of using GUIDE is that the function `guidata` cannot be used to manage any other variables apart from the handles structure. Therefore, application-defined data such as unit conversion constants are most easily shared among the callbacks by saving the data in new fields added to the handles structure.

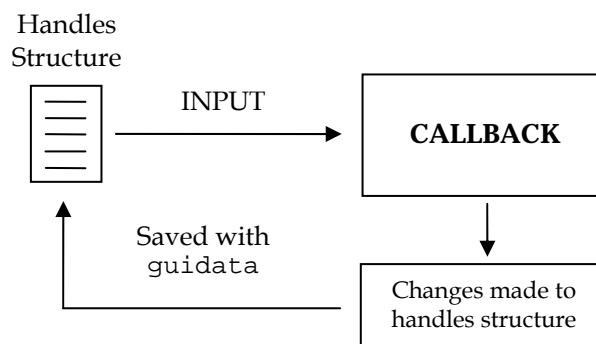


Figure 3: Block diagram illustrating how the handles structure is managed

Care must be taken if the callback calls another function that modifies data stored in the handles structure. Because the handles structure is not a global variable and each callback and function has its own workspace, multiple copies of the handles structure can exist at a given time in multiple workspaces. If `guidata` is called within the function, the updated version of the handles structure will not be visible to the callback. So to ensure that the handles structure is maintained correctly, it should be returned to a callback as an output from a function and `guidata` should only be called from the callback (Figure 4).

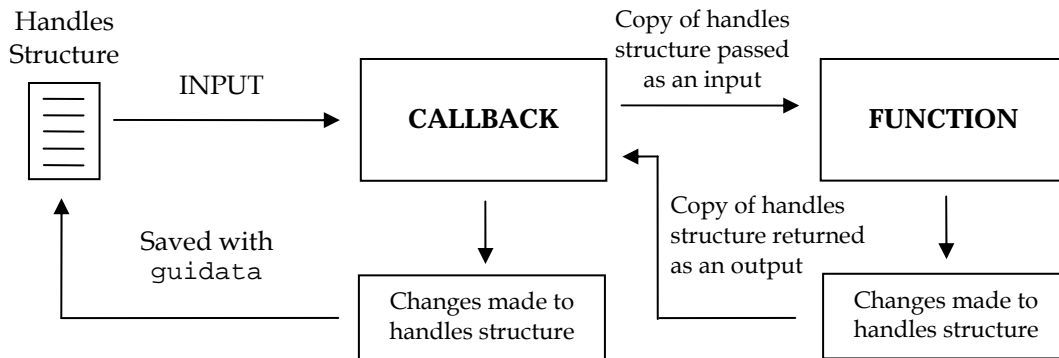


Figure 4: Block diagram illustrating how to correctly use the handles structure when a function is executed from within a callback

3.4 Displaying Data

Plots are used in the main window to display output data and show a preview of the platform flight path and sensor field of view. The panel in the bottom right corner of the main window allows the user to change the screen layout to one of the following combinations:

- A split-screen of the three-dimensional and top views of the scene.
- A split-screen of the three-dimensional views of the scene and output.
- A full screen three-dimensional view of the scene.
- A full screen three-dimensional view of the output.
- A split-screen of the three-dimensional views of the scene and output, and a top view of the scene.

3.4.1 Plotting the Scene

While MATLAB is a very powerful tool for displaying and manipulating data there are a number of issues that exist due to the size of the data used by the simulation. The biggest issues associated with displaying scene and output data are the memory required and processing time. A point cloud representation of a typical scene will consist of about four million voxels and creating a three-dimensional plot of this amount of data can take several minutes since it is necessary to assign a unique colour to each voxel based on its

height in the scene (z coordinate). Assigning these colours is required to give the impression of depth in a three-dimensional image displayed on a computer screen (Figure 5), but it does not happen automatically with point data as it does when generating a surface plot in MATLAB.

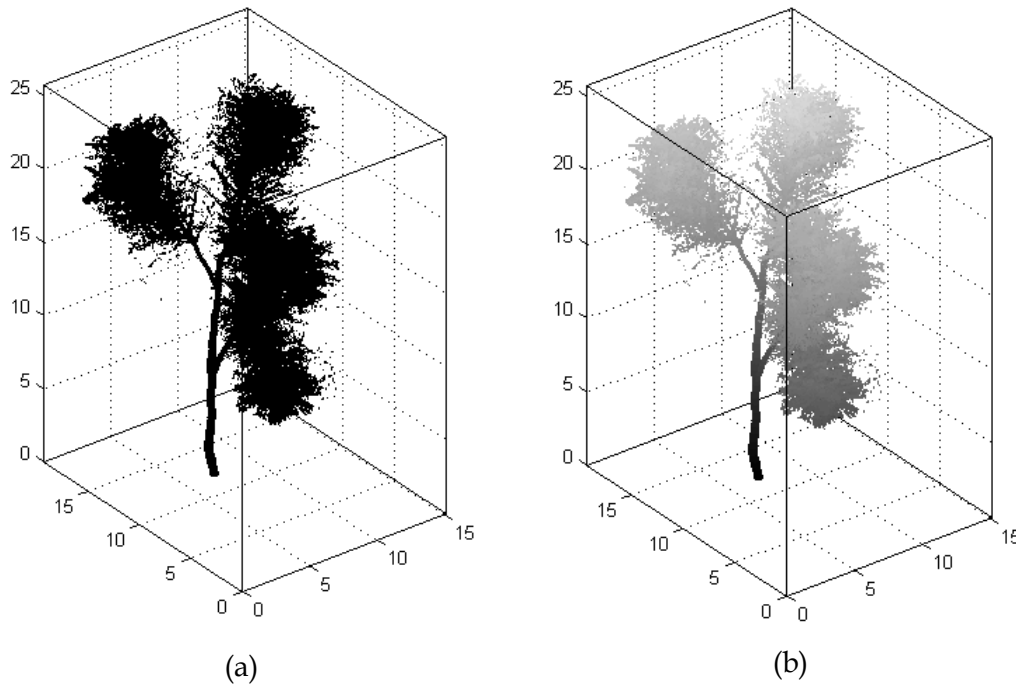


Figure 5: A single tree plotted in MATLAB using (a) one colour, and (b) shades of grey, which gives the impression of depth

Since there are a maximum of 64 colours available for plotting point data in MATLAB, the time taken to produce this three-dimensional plot can be reduced by grouping the data into blocks based on z coordinates and executing the `plot` command once for each colour instead of for each z value. This reduces the number of times the plot command needs to be called by a factor of ten. A gray scale consisting of 56 shades from near white to black is used to plot the scene data, which takes less than a minute for a scene consisting of millions of points.

3.4.2 Previewing the Flight Path

Part of the motivation behind developing this GUI was to show a preview of the aircraft's flight path and sensor's field of view at each scan location, so that an analyst can check that the scenario has been set up correctly before running the simulation. This preview is shown in both the three-dimensional and top-down views of the scene (Figure 6). The flight path and sensor field of view are updated without needing to replot the entire scene, via the handles to these two graphics objects which are stored in the handles structure (§3.3).

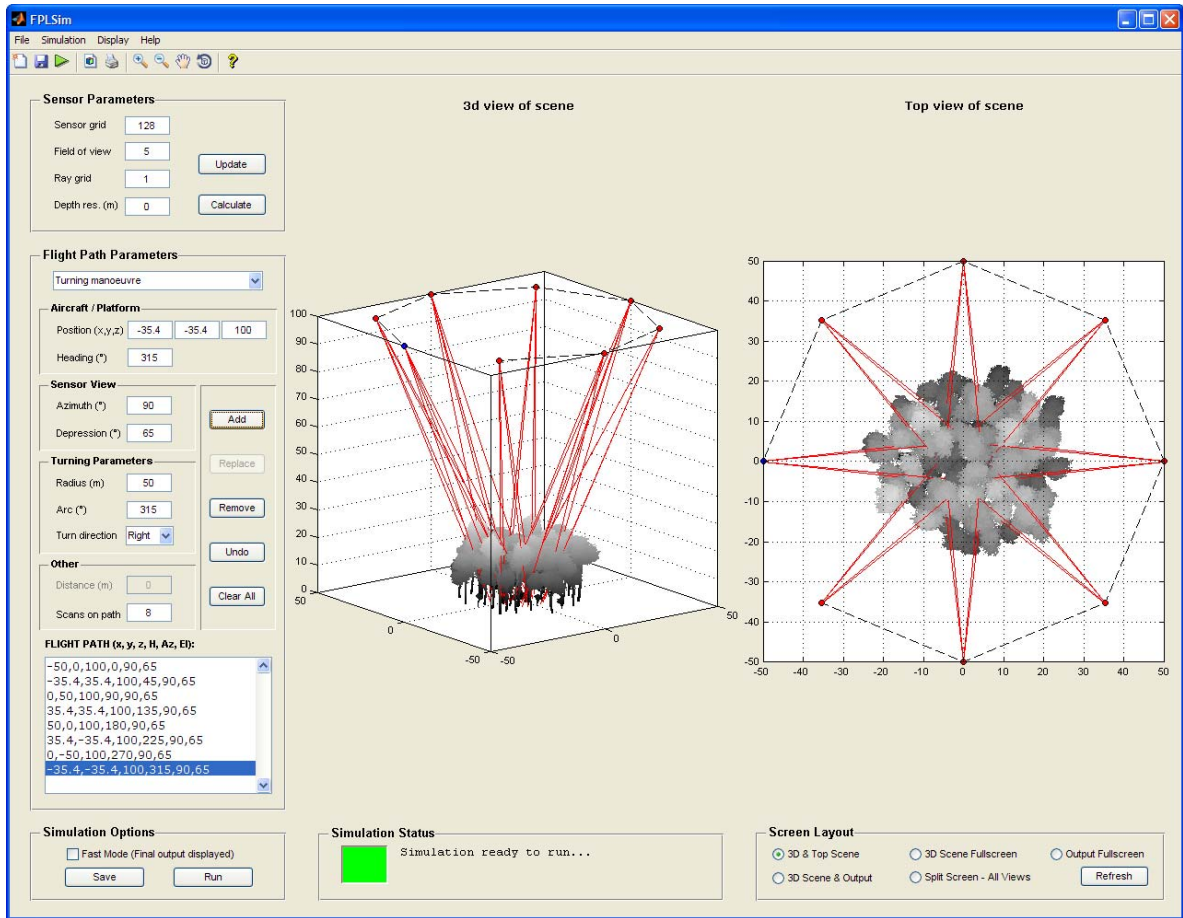


Figure 6: The main window of the LADAR simulation tool showing a flight path preview

3.5 Output Visualisation

The GUI includes MATLAB’s standard visualisation tools that allow a user to zoom, pan and rotate plots. These controls are located in the toolbar along the top of the main window and can be used with all views described in §3.4. The viewpoint for three-dimensional views can also be set manually by specifying azimuth and elevation angles in a dialog box which is accessed through the Display menu.

3.5.1 Limitations with Three-Dimensional Plots

One limitation with the current implementation of the zoom and pan controls (in MATLAB R2008b) is that they were designed for use with two-dimensional plots. When using these controls to change the view of a three-dimensional plot (such as the input scene or output data) unexpected results may occur such as the image extending off the screen because the axes are not rescaled, or the data being incorrectly clipped. To produce somewhat reasonable results, a workaround was discovered where the zoom control could be used from the top (x - y) view of a three-dimensional image. After performing the zoom, the image can be rotated as desired. This generally allows a section of a three-dimensional image to be zoomed in with the x and y axes correctly rescaled and clipped.

3.5.2 Applying a Threshold to the Output

Functionality has been included in the GUI to allow a height threshold to be applied to the output image (Figure 7). This is particularly useful in foliage penetration where we are interested in objects beneath a partially obscured area of interest. This option is available from the Display menu. When a user specifies a height (in metres), the output figure is replotted up to this height.

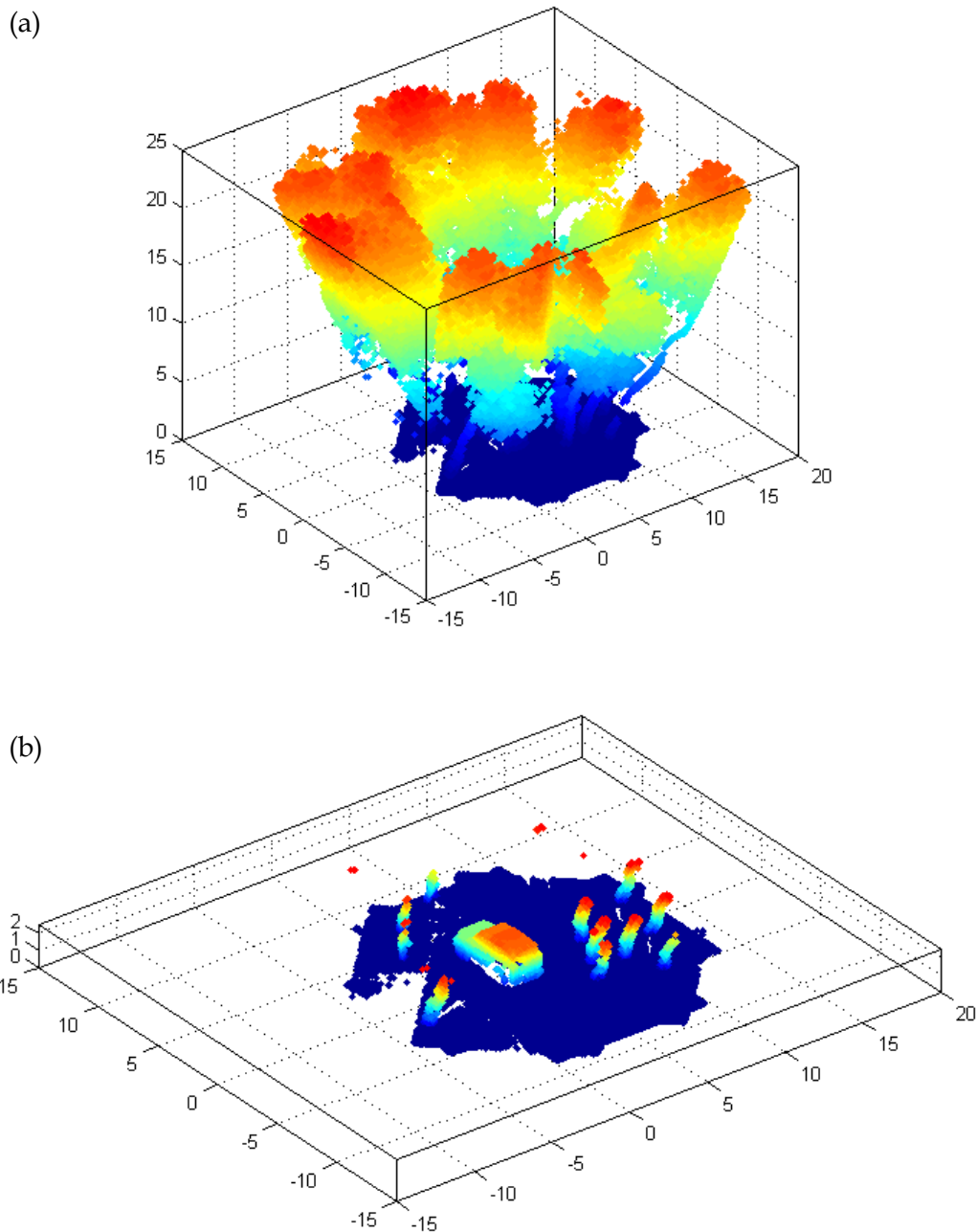


Figure 7: (a) Example of a simulation output before applying a height threshold. (b) Simulation output after a height threshold of 2.5 m has been applied

4. Program Overview

The LADAR simulation tool consists of four stages: File Setup, Parameters, Simulation and Output. The flow of the program and the user actions required through each of these stages is illustrated in Figure 8. The program is designed so that the user will step sequentially through each of these stages. However, users can go back to modify actions performed in a previous stage at any time, although this may have an impact on the subsequent stages.

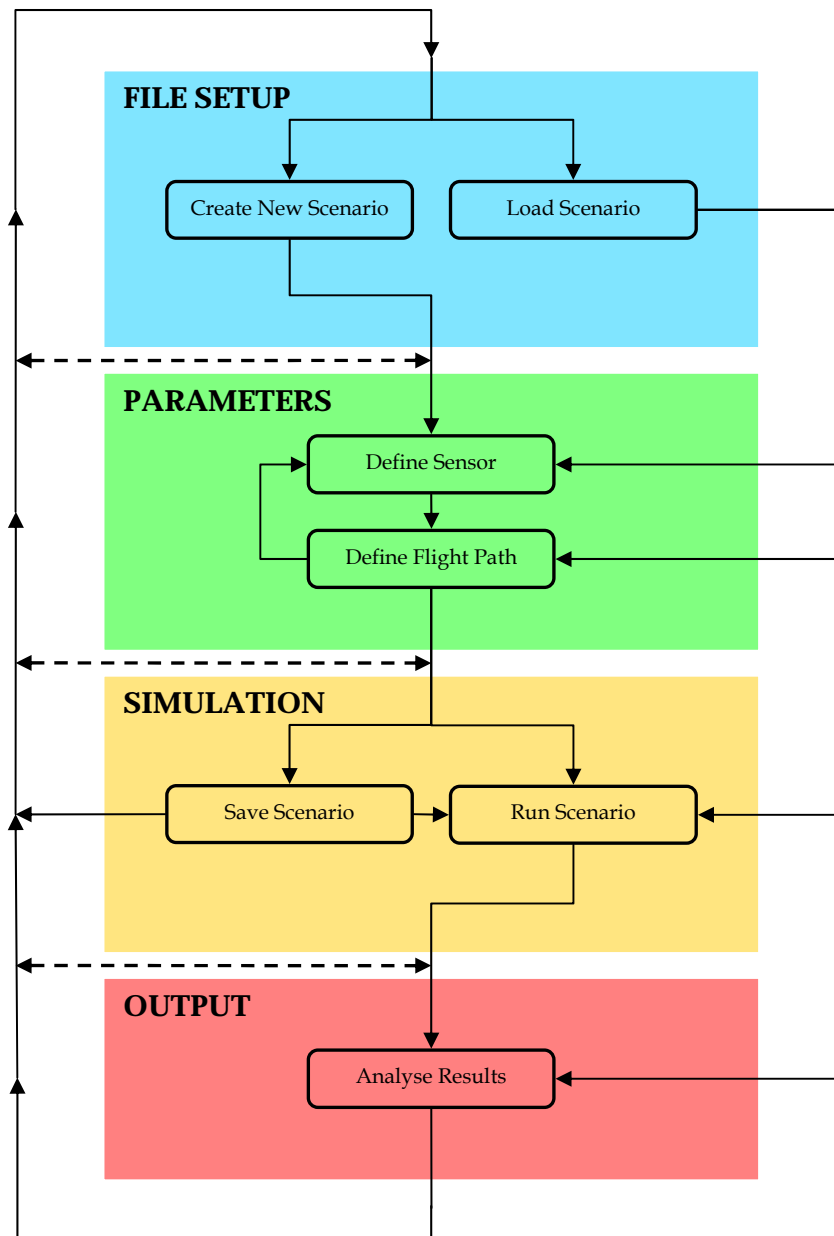


Figure 8: Flow diagram of the program showing the user actions that are performed at each stage

4.1 File Setup

The first stage of the program involves setting up a scenario (Figure 9). There are two options available to the user, either setup a new scenario or load an existing scenario.

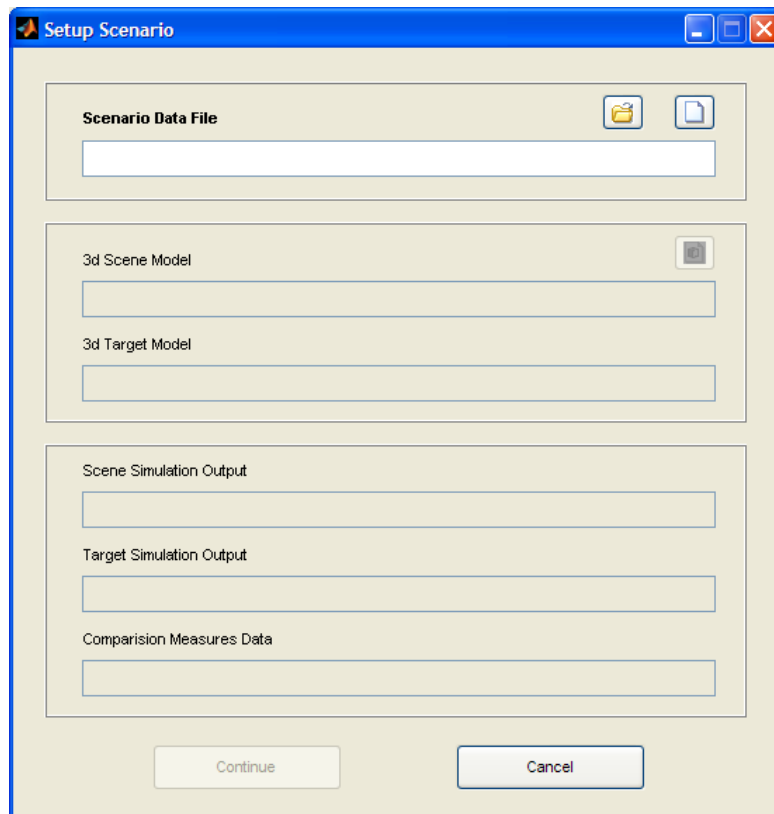


Figure 9: The setup scenario dialog window

For a new scenario the user is required to create a new scenario data file and load the three-dimensional scene model file. The scenario data file will contain file information, sensor and flight path parameters. The scenario data filename will also be used to automatically generate filenames for the output and comparison measures data.

For an existing scenario the user is required to specify the scenario data file to load. Checks are performed at this stage to ensure all associated output and model files exist. If a file is not found, the scenario cannot be loaded and an error message will be displayed. An overview of the different files used by the program is given in Table 2.

Table 2: Description of files used by the program

| File | Description |
|--------------------------|--|
| Scenario Data File | A text file with a .dat extension (data file), containing Scene and Target model filenames and the sensor and flight path parameters. |
| 3D Scene Model | A text file (.txt) containing the information used to construct a three-dimensional model of the scene. This information is represented as a list ($n \times 3$ array) of locations stored as comma-delimited values. These locations represent the spatial information of elements, e.g. trees and targets, in the scene. |
| 3D Target Model | A text file (.txt) in the same format as the scene model file, containing the information used to construct a three-dimensional model of the target only (unobscured by foliage). This single target is positioned in the same x - y location as in the scene model. The filename of the target model is linked to the scene model, i.e. if the scene model is <i>ExampleScene.txt</i> then the corresponding target model filename will be <i>ExampleScene_Target.txt</i> . |
| Scene Simulation Output | A text file (.txt) in the same format as the model files, containing scene simulation output (results of running the simulation with the scene where the target is obscured by foliage). The filename of the scene simulation output is linked to the scenario data file, i.e. if the scenario data file is <i>ExampleScenario.dat</i> then the scene simulation output filename will be <i>ExampleScenario_Scene.txt</i> . |
| Target Simulation Output | A text file (.txt) in the same format as the model files, containing target simulation output (results of running the simulation with the target only, unobscured by foliage). The filename of the target simulation output is linked to the scenario data file, i.e. if the scenario data file is <i>ExampleScenario.dat</i> then the target simulation output filename will be <i>ExampleScenario_Target.txt</i> . |
| Comparison Measures Data | A text file with a .cmf extension, containing the results of some comparison measures computed on the output data. The filename of the comparison measures data is linked to the scenario data file, i.e. if the scenario data file is <i>ExampleScenario.dat</i> then the comparison measures data filename will be <i>ExampleScenario.cmf</i> |

4.2 Defining Parameters

After the file setup is complete and either a new scenario has been created or an existing scenario has been loaded, the user can define the parameters for the flight path (§6) and sensor (§7) using the controls located in the associated panels on the left side of the main window (Figure 2). If an existing scenario has been loaded, the parameters stored in the data file will be automatically loaded into the appropriate parameter textbox fields.

4.3 Running and Saving the Simulation

After a scenario has been set up (i.e. scenario data file created or loaded, model data files loaded, and sensor and flight path parameters defined), the simulation is ready to run. At this stage it is possible to save the scenario data file information to a new file via the ‘Save’ push button in the Simulation Options panel (Figure 2). The simulation is started by selecting ‘Run Simulation’ either from the Simulation menu, the toolbar or the ‘Run’ push button in the Simulation Options panel.

While a simulation is running:

- all menus, toolbars and buttons are disabled to prevent other actions being performed;
- the progress is displayed in the simulation status panel; and
- the output plot is progressively updated unless the 'Fast Mode' option has been selected by enabling the check box in the Simulation Options panel, in which case only the final output will be displayed.

Suppressing progressive plotting by selecting 'Fast Mode' significantly improves the processing time, particularly for large simulations where the scene model contains millions of voxels.

4.4 Comparing the Output

When a simulation run is completed, the output is automatically saved to the filename defined when the scenario was initially set up. The software will now perform calculations using the output results to compute some comparison measures (discussed in §2) for the unobscured and obscured views of a single target. Simultaneous comparison of multiple targets in a scene is currently not implemented in the software. These measures are only computed for the target that is saved in the '3D Target Model' file. A dialog window (Figure 10) will be displayed with a selection of these results for a quick comparison between different scenarios.

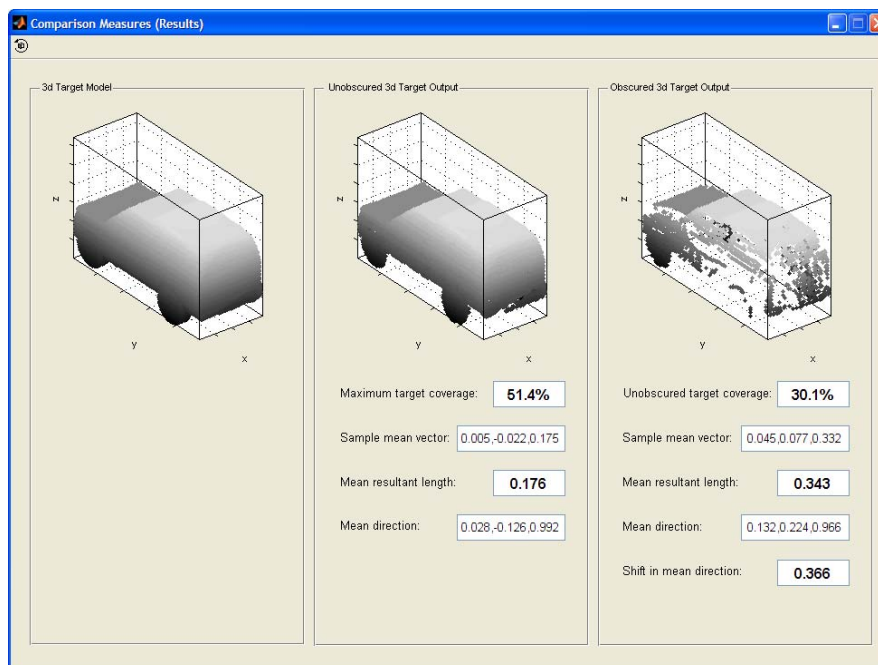


Figure 10: Example of the comparison measures dialog window, displayed after running the simulation. The measures displayed here are only preliminary and the measures actually used in future analysis work may be different.

5. Scene Model Data

The three-dimensional models of the scene data are constructed from locations stored in a text file. These locations represent voxels (three-dimensional volume elements), which form the spatial information of the elements contained in the scene.

Two methods were investigated for storing the model data:

- Use a three-dimensional array in MATLAB to represent all possible voxels, with ones and zeros indicating whether the voxel contains anything or not. This was the simplest option but is impractical since a realistic scene is too large to be stored and processed in this way on a single computer.
- Store the data using a list and omit voxels representing empty space in the scene. This eliminates the issues with memory but working with the data in this form is more complicated and so has a significant impact on processing time.

The following example gives a comparison of the memory requirements of both methods and illustrates the benefit of storing the data using a list as opposed to using a three-dimensional array. Figure 11 shows a typical scene made up of trees and vehicle. This model contains about 3.8 million voxels representing the scene inside a volume of approximately $1500 \times 1500 \times 900$ voxels.

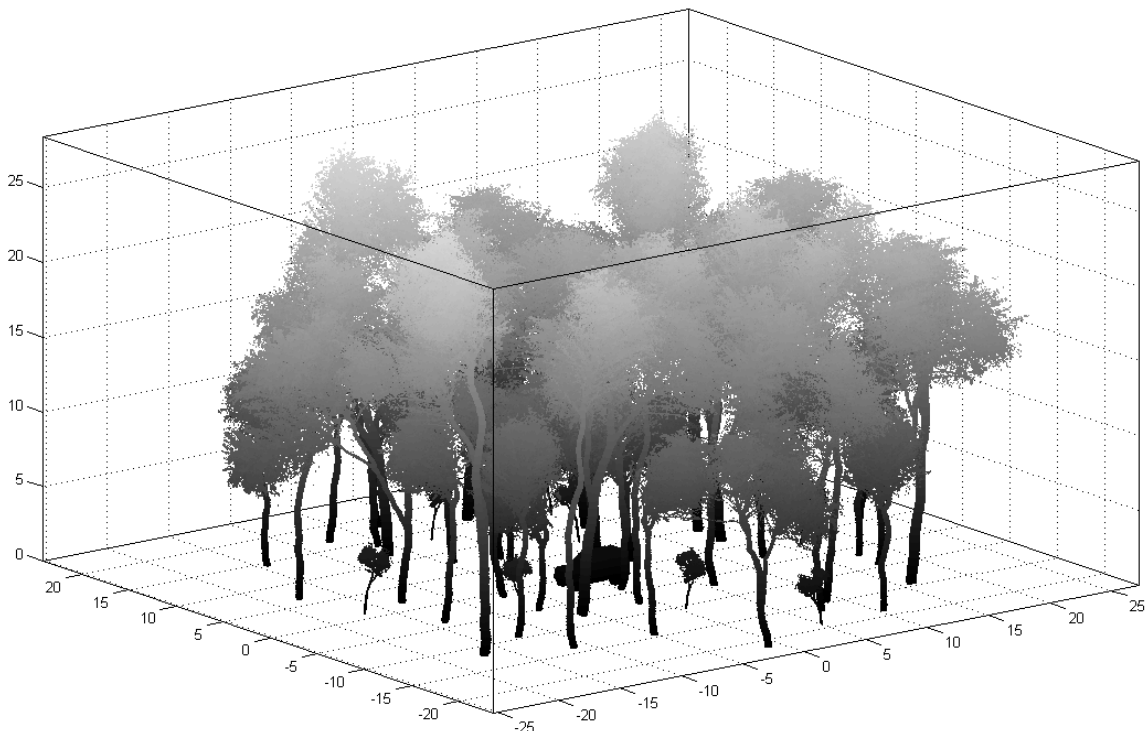


Figure 11: A three-dimensional model of a scene consisting of trees and a vehicle

Storing the data in a three-dimensional array would require an array the size of $1500 \times 1500 \times 900$, and 1 byte of memory is required to store a value of 1 or 0 in each element of the array using the unsigned 8-bit integer data type in MATLAB. Therefore the three-dimensional array method will require approximately 1.9 GB of memory. Alternatively, if a list is used to store only the voxels that represent elements in the scene, an array the size of $3.8 \times 10^6 \times 3$ is required. Here each row will contain the (x,y,z) location of a single voxel, and 8 bytes of memory is required to store the coordinate value in each element of the array using the double data type in MATLAB. Therefore the list method will require approximately 87 MB of memory. This method is much more efficient since less than 1% of all voxels form elements in the scene.

Using the list method is similar to MATLAB sparse matrices, although they could not be used for this application since they are restricted to two-dimensional arrays. Although it has been demonstrated that the list method overcomes the memory issues associated with a three-dimensional array it has other implications for working with the data, as described in §7.3.

6. Flight Path Modelling

The flight path modelled in the simulation is represented as a sequence of scan locations defined by position and heading parameters set by the user. The position (which refers to the sensor position on the aircraft) is given by (x,y,z) coordinates expressed in metres, where positive x corresponds to east, positive y corresponds to north, and positive z corresponds to altitude. The heading is the aircraft's direction of travel, measured in degrees clockwise from north.

Defining sensor scanning positions also requires the sensor pointing direction (§7.2). This is defined by azimuth and depression angles. The sensor azimuth is measured in degrees clockwise from the nose of the aircraft, i.e. the sensor azimuth is 0° when the sensor is pointed straight ahead. The sensor depression is defined as the angle from horizontal, with a range of 1° to 90° where 90° is nadir.

Sensor scanning positions can be defined by the user by manually setting each of the four parameters: aircraft position, aircraft heading, sensor azimuth and depression. However, two manoeuvres have been defined to simplify and automate the modelling of an aircraft flight path:

- The aircraft travels in a straight line and maintains its current heading (Figure 12). This manoeuvre is defined using the aircraft's starting position, the distance (in metres) to travel along the current heading from the starting position, and the number of scanning positions to be placed at evenly spaced intervals along this path.
- A turning manoeuvre (Figure 13) defined by the aircraft's starting position, the turning radius (in metres), the arc (angle of the turn, in degrees), direction (right or left) and the number of scanning positions to be placed at evenly spaced intervals along the turn.

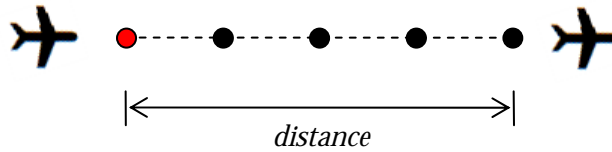


Figure 12: Defining a straight line manoeuvre with five scans along the path. The starting position of the aircraft is shown in red.

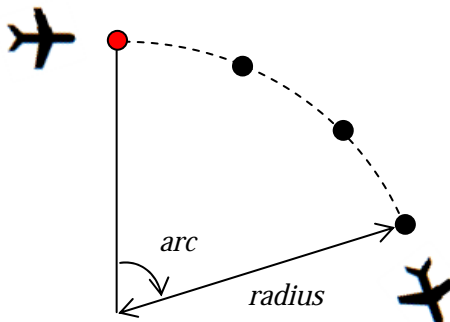


Figure 13: Defining a right turn manoeuvre with four scans along the arc. The starting position of the aircraft is shown in red.

It is possible to create detailed flight paths for a scenario using a combination of these manoeuvres along with manual adjustments as required.

7. Sensor Modelling

A low fidelity sensor model was implemented in the simulation since the purpose of developing the software was to analyse how to best task the aircraft and position the sensor rather than developing a high fidelity model for simulating the sensor itself. The modelled sensor does not consider factors such as noise, probability of detection, probability of false alarm, detector crosstalk, or errors due to timing circuit resolution. These random effects were not modelled since it was decided that a more meaningful comparison of the flight paths could be made without them.

The output of the sensor modelling is the set of voxels that are within the sensor's field of view and have direct line-of-sight to the sensor, rather than a reconstruction of the correct angle-angle-range data that such a real sensor would actually use to produce an image.

7.1 Defining the Sensor

The sensor model implemented in the software is defined by the following parameters which are set by the user:

- **Sensor grid** – defines the pixel resolution of the sensor, which is assumed to be a square array. If the user enters value n , then the sensor consists of an $n \times n$ array of detectors.
- **Field of view** – this is the angular extent of the scene that the sensor can view at any given moment and is specified in degrees.
- **Ray grid** – controls the number of rays that are received by each detector. If the user enters value m , then the ray grid consists of an $m \times m$ array. This represents the number of different angles from which photons could arrive at the detector array.
- **Depth resolution** – this is determined by the response time of the detector (the time it takes for the detector to reset after receiving a photon, typically in the order of nanoseconds). The depth resolution is specified as a distance in metres, i.e. the detector response time multiplied by the speed of light. Multiple voxels with line-of-sight to a given detector must be separated by at least this distance in order to all be detected.

7.2 Setting-up the Ray Data

To simulate the pulse of laser light reflected back to the detectors, rays are defined to trace from a sensor grid and into the three-dimensional model. The output image is then formed from the voxels intercepted by the rays.

A ray is defined by:

1. *Ray position*, which is the (x,y,z) floating-point coordinates of some point along the ray. An initial ray position is the starting point for a ray.
2. *Tracing vector*, which sets the direction the ray will trace into the scene. The tracing vectors are defined under the assumption that they are equally spaced in angle and do not cross.

A set of rays are initially defined for a downwards-looking sensor located at the origin of the (x,y,z) space (Figure 14a), which is then rotated and translated as required (Figure 14b-c). The tracing vectors and initial ray positions are calculated using the sensor grid, ray grid, field of view and the pixel pitch of the sensor. The pixel pitch is hard-coded in the program as $100 \mu\text{m}$ [1].

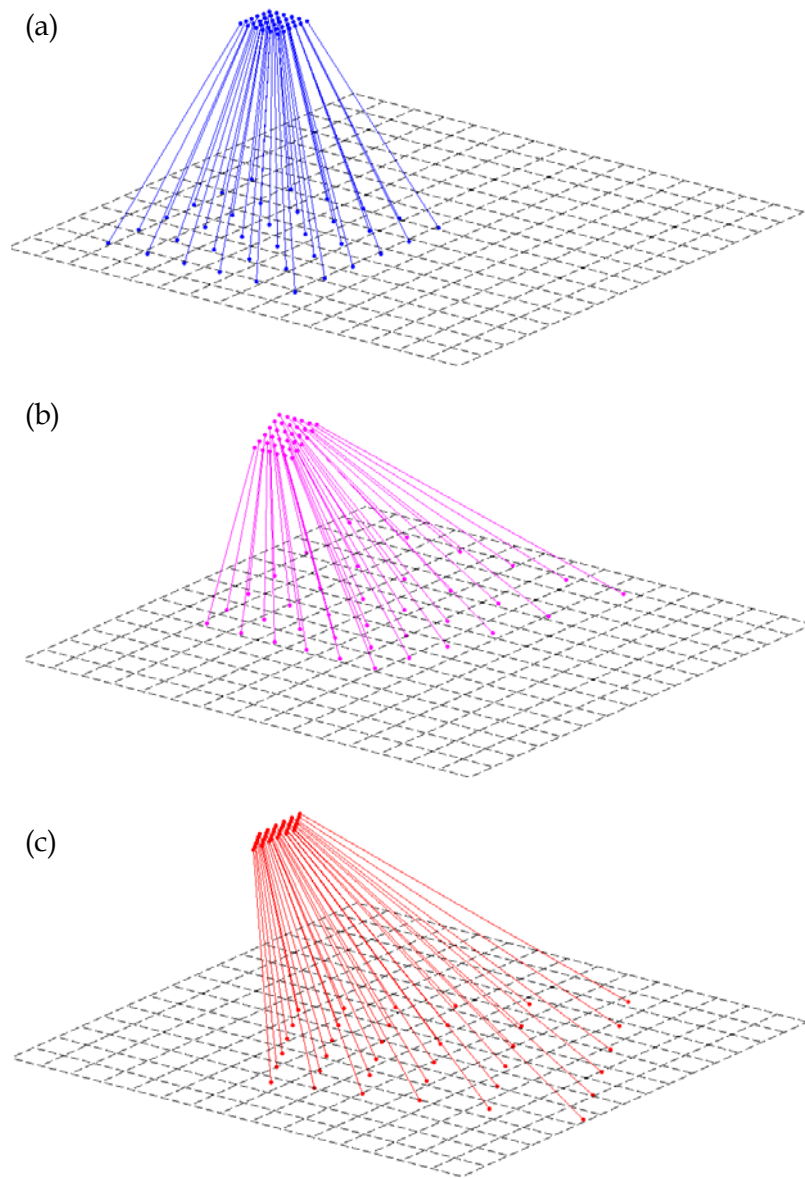


Figure 14: A set of rays defined by a 3×3 sensor grid, 2×2 ray grid, and 45° field of view. (a) Defined about the origin of the (x,y,z) space. (b) Orientation after applying a sensor depression angle of 60° . (c) Final orientation after applying a sensor azimuth of 40° and aircraft heading of 20° .

The sensor's pointing direction depends on the aircraft position and heading, and the sensor azimuth and depression angles. The first step is to tilt the sensor by the specified depression angle D (rotation about the x axis), by multiplying the ray positions and tracing vectors (which define the path of each ray) respectively by the rotation matrix, R_x (Figure 14a-b):

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(90^\circ - D) & -\sin(90^\circ - D) \\ 0 & \sin(90^\circ - D) & \cos(90^\circ - D) \end{bmatrix} \quad (1)$$

The next step is to rotate the sensor in azimuth (about the z axis) by the sum of the aircraft heading H and the sensor azimuth A , using the rotation matrix R_z (Figure 14b-c):

$$R_z = \begin{bmatrix} \cos(-(H+A)) & -\sin(-(H+A)) & 0 \\ \sin(-(H+A)) & \cos(-(H+A)) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (2)$$

The final stage of setting up the ray data to represent the sensor is to translate the centre of the grid of initial ray positions from the origin of the (x,y,z) space to the aircraft's position. The ray positions and tracing vectors are now passed into the ray tracing algorithm.

7.3 Preparation for Ray Tracing

Ray tracing is conceptually very simple: step along a tracing vector until the ray either enters a voxel that forms part of the scene or reaches the ground. Ray tracing therefore involves repeatedly searching the list of locations forming the model of the scene.

7.3.1 Defining the Search Space

A search space, i.e. a subset of the total list of locations, needs to be defined to reduce the computation time. This is a consequence of choosing to store the three-dimensional scene using a list of locations as opposed to using a three-dimensional array. If it were possible to store the scene data directly in a three-dimensional array, then it would not be necessary to define a search space since the ray tracing algorithm could directly reference a voxel that it passes through and determine whether it is empty or not. However, using a list of locations means that a search is required to determine if a ray has intercepted part of an element in the scene. Since the three-dimensional scene consists of millions of voxels, searching potentially the entire list for every single voxel that every ray passes through is impractical. Therefore before performing the ray tracing, a search space is defined to restrict the search first by z value and then by x - y values.

7.3.2 Restricting by z Value: Layers

The program has been designed to simulate the specific case of an airborne sensor used to search for ground targets. As a result the z component of the tracing vector is always negative, i.e. the z coordinate (altitude) of the ray position will always decrease as a ray is traced. Therefore the list of locations is first sorted by decreasing z coordinate value and then (arbitrarily) by increasing y and increasing x values. The highest locations that make up the scene will now be at the top of the list and the locations on the ground will be at the bottom of the list. Note that if a horizontally-looking land-based sensor was being simulated then the list of locations could instead be sorted using the x or y values.

The sorted list of locations, which we will call *Mdata*, can now be pre-searched to determine start and end pointers for each subset of data with the same z coordinate. We will refer to such subsets as *layers*, and store the pointers in the array *zpoint*. So when searching for the voxel containing the ray position in the list of locations, the z coordinate

of the current ray position can reference *zpoint* to constrain the search to the appropriate layer. Ideally, the index of *zpoint* would refer directly to the *z* coordinate of *Mdata*. But because MATLAB starts indexing at 1 instead of 0, it is necessary to include an offset so that the first value of *zpoint* refers to location [0,0,0].

The start and end indices defining layer $z = z_n$ in *Mdata* are inferred from *zpoint* as:

$$start_point(z_n) = zpoint(z_n + 1) \tag{3}$$

$$end_point(z_n) = zpoint(z_n) - 1 \tag{4}$$

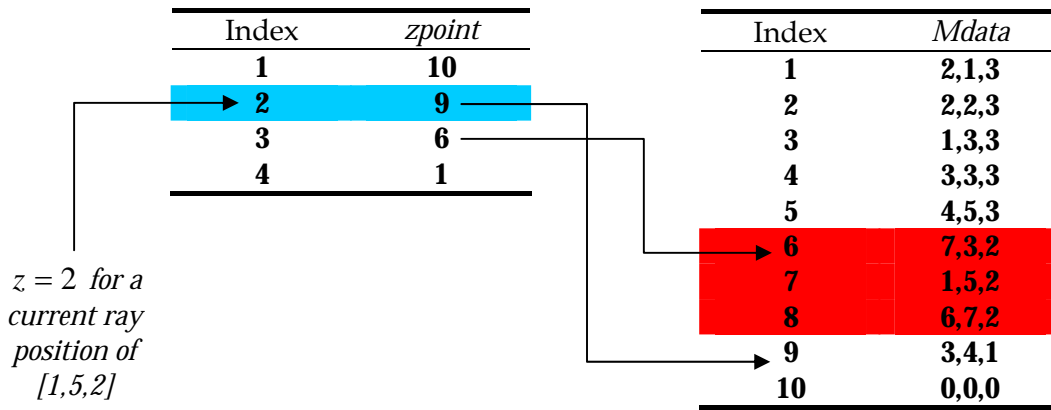


Figure 15: Example showing how pointers are used to define the search space. The search space for a current ray position of [1,5,2] is shown in red. This is inferred from *zpoint* (shown in blue for this example) which gives the start and end pointers of *Mdata*.

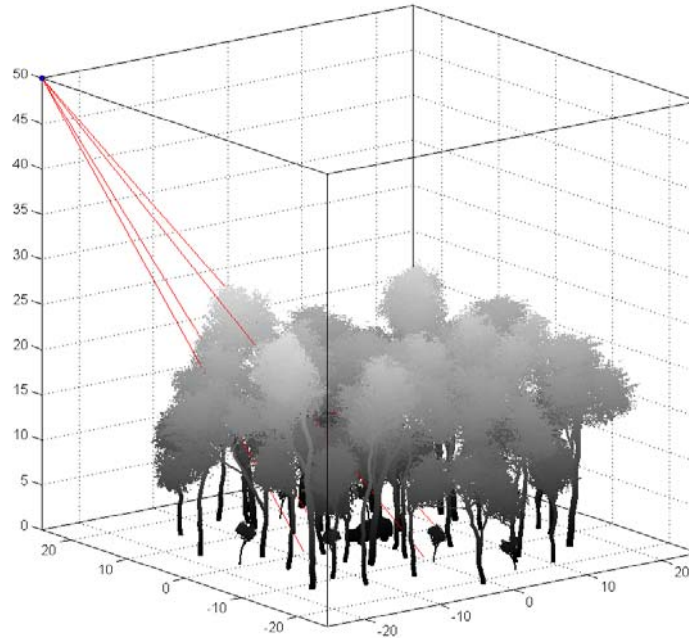
This process is illustrated in Figure 15 with some sample data where the current ray position is [1,5,2]. Since the *z* coordinate is 2, the start pointer is $zpoint(2+1) = 6$ and the end pointer is $zpoint(2) - 1 = 8$. Therefore the *Mdata* list needs only to be searched from indices 6 through to 8 inclusive. In this case, the search will terminate when it checks index 7 of *Mdata* because the position matches a location in the list indicating an intercept.

The final point to note is that the last location in the data is always the origin [0,0,0]. While not actually a location in the three-dimensional model data, since ground level (or lowest point if the scene contains landforms) is defined as $z=1$, the origin serves as a ‘dummy’ location so that the end pointer of the last *z* coordinate in the list can be referenced.

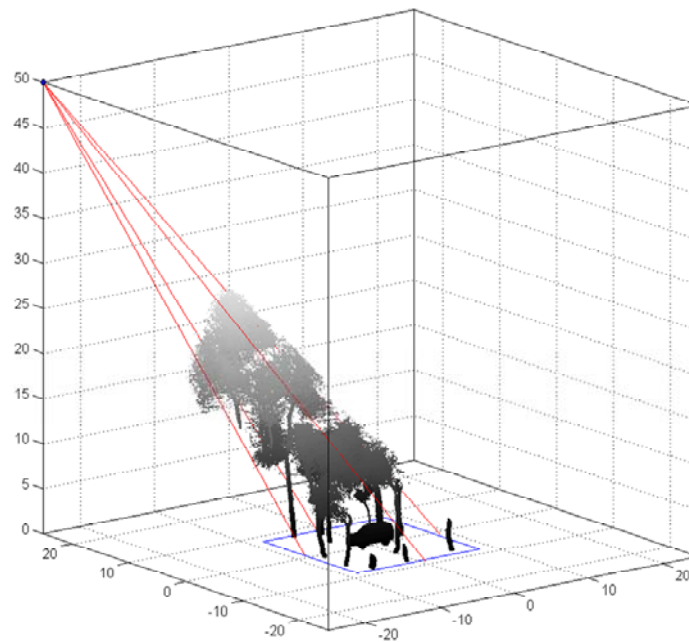
7.3.3 Restricting by *x-y* Values: Bounding Boxes

While using pointers to restrict the search to the appropriate layer has a significant improvement in the computation time, the speed of a search can be further improved by considering only the region of each layer that is within the field of view of the sensor. This can be approximated by a *bounding box* defined by the maximum and minimum *x* and *y* values in the field of view on each layer (Figure 16). While the bounding box is generally a larger area than the strict field of view, it is much simpler to determine since only the

intercepts of the four corner rays with each layer need to be calculated. It is also easier to restrict the data to a rectangle aligned with the x - y grid than the arbitrary trapezium shape of the strict field of view.



(a)



(b)

Figure 16: Pruning the list of locations to improve the speed of the search. The field of view is shown in red. The blue square shows the field of view aligned with the x - y grid on layer $z=1$. (a) Plot of the original scene, $Mdata$. (b) Plot of the pruned list, $Bounded_Mdata$, used for searching.

The coordinates defining the bounding box are used to prune the original list ($Mdata$), and form a new list ($Bounded_Mdata$). At the same time a new set of pointers ($Bounded_zpoint$) is computed for this pruned list, allowing the appropriate layer to be searched as described earlier but with the search restricted to the bounding box.

7.3.4 Multiple Bounding Boxes

The computation time can be significantly improved further if instead of applying a single bounding box, the sensor is divided up and bounding boxes are computed for subsets of the rays. The bounding box (and z layer) defines the search space for a given subset of rays. Using a larger number of smaller bounding boxes means that there will be fewer voxels in the search space for a given ray, which reduces the time spent searching for voxels in the ray tracing algorithm even though the bounding boxes will in general overlap. However, the time taken to define a large number of bounding boxes will eventually offset the gains made in reducing the search time. The optimal number of bounding boxes depends on the number of rays to be traced (Figure 19).

The total computation time is very specific to a particular scenario and the computer used to run it. However, the times taken by the bounding box and ray tracing algorithms have been modelled as a function of the number of voxels per bounding box for the input scene shown in Figure 11, which contains approximately 3.8 million voxels. These times were collected from test scenarios run using MATLAB R2008b on a HP Compaq 8710w laptop computer with an Intel Core 2 Duo CPU T9500 2.6 GHz, 4.0 GB RAM and Microsoft Windows XP Professional. The plot of the time taken versus number of voxels contained in a bounding box, v , for the bounding box (Figure 17) and ray tracing (Figure 18) algorithms show that a linear model with a non-polynomial (logarithmic) term provides a good fit of the data.

The bounding box algorithm time, B_t , is modelled by:

$$B_t = a_0 + a_1 v + a_2 v \ln(v) \quad (5)$$

$$\text{where } a_0 = 0.20, a_1 = 1.25 \times 10^{-6}, a_2 = -7.31 \times 10^{-8}$$

The ray tracing algorithm time (for a single ray), R_t , is modelled by:

$$R_t = b_0 + b_1 v + b_2 v \ln(v) \quad (6)$$

$$\text{where } b_0 = 0.001, b_1 = 5.08 \times 10^{-8}, b_2 = -2.81 \times 10^{-9}$$

Equations (5) and (6) are not particularly helpful in choosing the appropriate number of bounding boxes that will minimise the computation time for a given scenario because it is difficult to estimate the number of voxels that will be contained inside the bounding boxes. The number of voxels contained inside a bounding box is dependent on the sensor pointing direction, the sensor's field of view, the density and distribution of voxels that form a particular scene and the size of the bounding box itself.

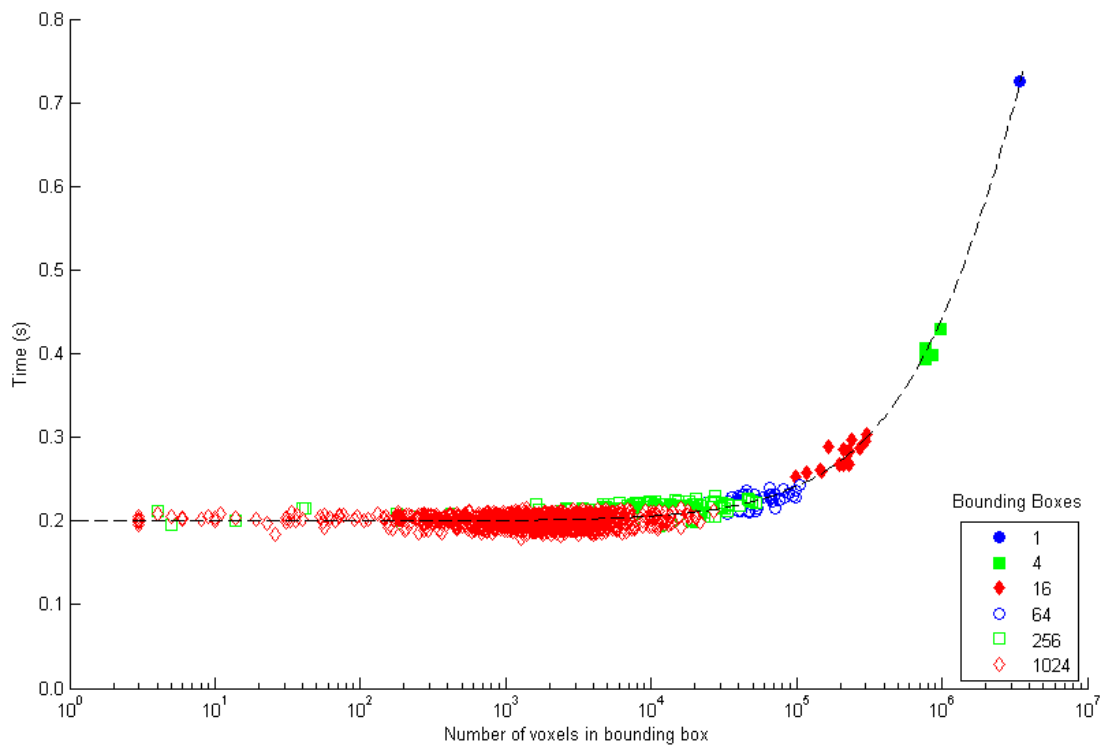


Figure 17: Time taken by the bounding box algorithm as a function of the number of voxels contained in a bounding box

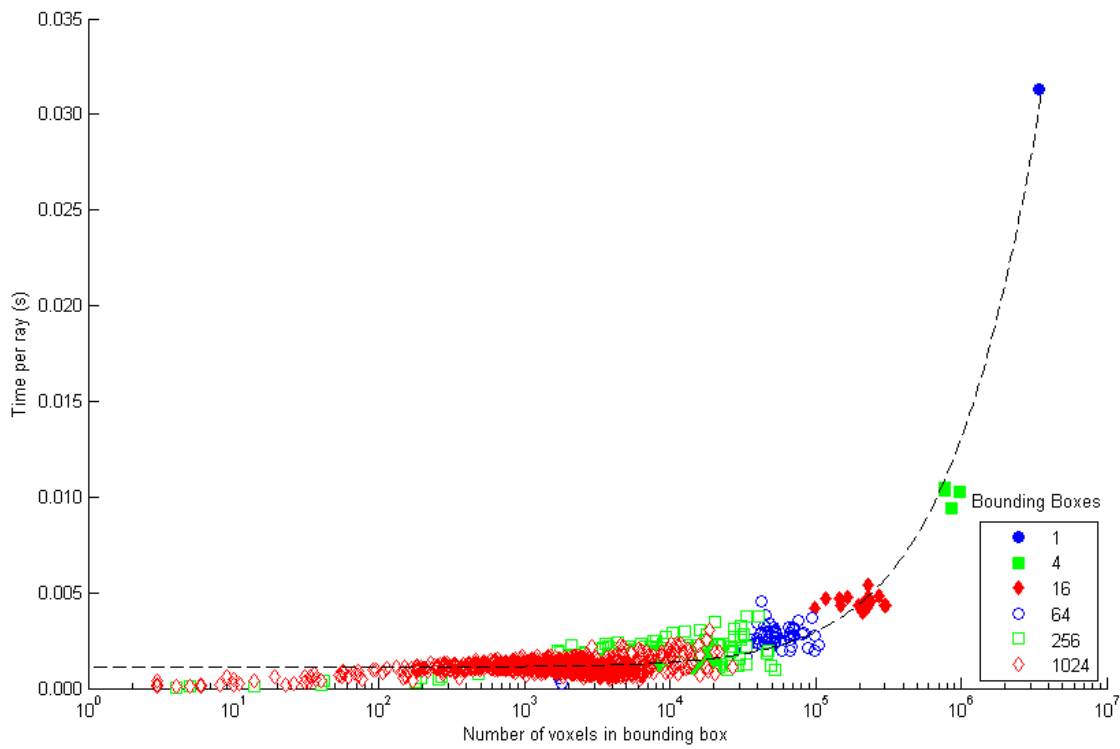


Figure 18: Average time taken by the ray tracing algorithm (for a single ray) as a function of the number of voxels contained in a bounding box

For scenarios using the input scene shown in Figure 11 and where the sensor azimuth angle is aligned with the x - y plane, the number of voxels contained in a bounding box can be approximated by the number of voxels contained in a single bounding box encapsulating the entire field of view, divided by the number of bounding boxes.

The total computation time, T , for a given number of bounding boxes, B , is estimated by:

$$T = B \cdot B_t(v) + N \cdot R_t(v) \quad (7)$$

where N is the number of rays and the number of voxels per bounding box is $v = V/B$, where V is the total number of voxels contained in a single bounding box encapsulating the entire field of view.

Figure 19 shows the times estimated by Equation (7) for a realistic scenario consisting of a 128×128 sensor array with 1, 4 and 16 rays per detector, i.e. 16384, 65536 and 262144 rays respectively. These estimates are accurate when the sensor azimuth angle is closely aligned with the x - y plane and the sensor depression angle is close to nadir. When the sensor is rotated and/or tilted, the size and overlap of the bounding boxes will increase and thus Equation (7) will tend to underestimate the computation time. However, the location of the minimum of the curve does not shift greatly. Since the curves are relatively flat around their minima, choosing a number of bounding boxes that does not correspond to the exact minimum will have only a small impact on the total computation time.

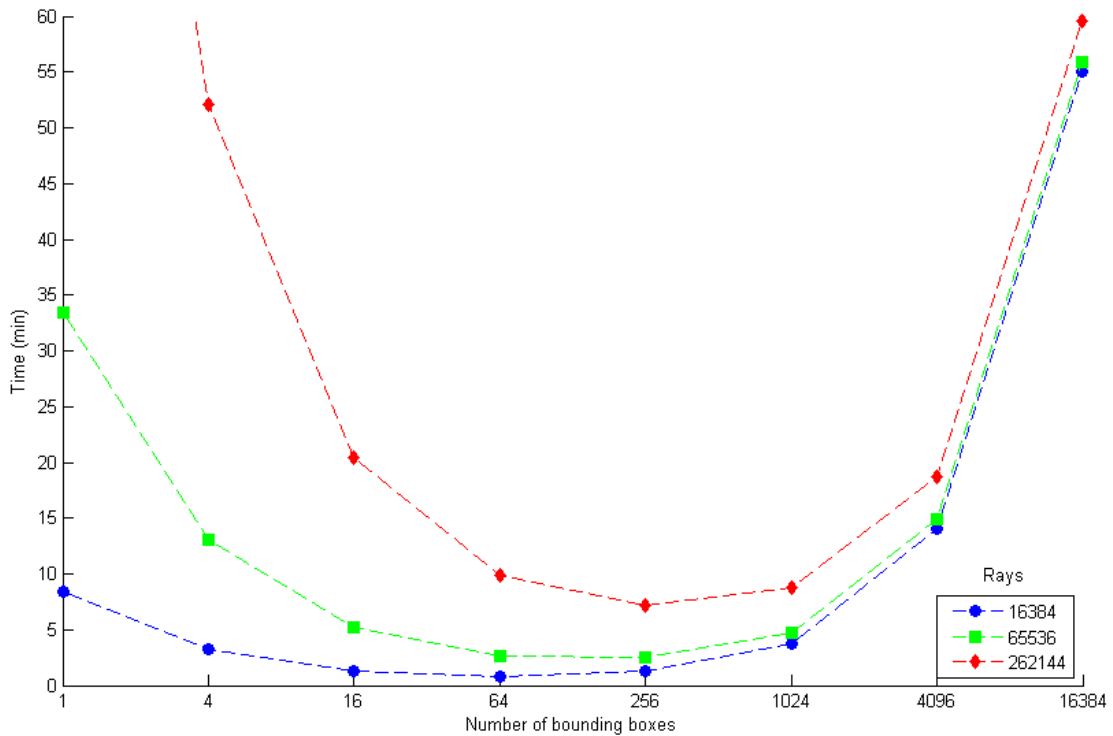


Figure 19: Approximate computation time of an example scenario (where the sensor depression angle is nadir and the field of view encapsulates the entire scene) for different choices of the number of rays traced and bounding boxes

7.4 Ray Tracing Algorithm

As previously mentioned, ray tracing is simply stepping along a tracing vector until the ray either enters a voxel that forms part of the scene or reaches the ground.

The ray tracing algorithm requires three inputs:

1. The ray data which includes the set of ray positions and associated tracing vectors.
2. The three-dimensional model data (*Bounded_Mdata*). This is the pruned list that only contains a subset of the model data around the field of view of the set of rays.
3. The array of pointers (*Bounded_zpoint*) used to restrict the search to a smaller subset of the pruned model data list.

The algorithm for tracing a single ray is illustrated in Figure 20. This algorithm runs in a loop to trace all the rays used to model the sensor in a sequential manner. To avoid unnecessary searching of the model data and so improve computation time, the ray tracing starts from the highest layer in the scene (i.e. top of the tree canopy) instead of the actual sensor. The position where a ray will intercept the highest layer is calculated using its initial position, tracing vector and the model data.

As a ray traces through the three-dimensional model, the current voxel location is determined by rounding the components of the ray's position to the nearest integers. However, the ray's position is maintained as a floating point value and not reset to the current voxel location as it continues to trace through the scene. This current voxel location is checked against the locations of voxels forming the three-dimensional model. If there is a match, the ray has intercepted part of an element in the model.

Two alternative methods were devised for performing this intercept check:

1. **Search Method** – A search is performed to determine if the ray has intercepted a location within the bounds of the current layer.
2. **Logical Indexing Method** – A logical indexing operation is used to check for an intercept within the bounds of the current layer. This method is currently implemented since it is significantly faster than the Search Method (tests have shown an approximate five times speed-up).

The ray tracing algorithm returns the intercepted locations or the positions where rays hit the assumed ground layer if they trace through the scene without an intercept. To apply the depth resolution parameter, the returned locations are first grouped by the detectors to which they correspond. Some returns may then be discarded so that the remaining returns are spaced appropriately based on the time to reset the detector.

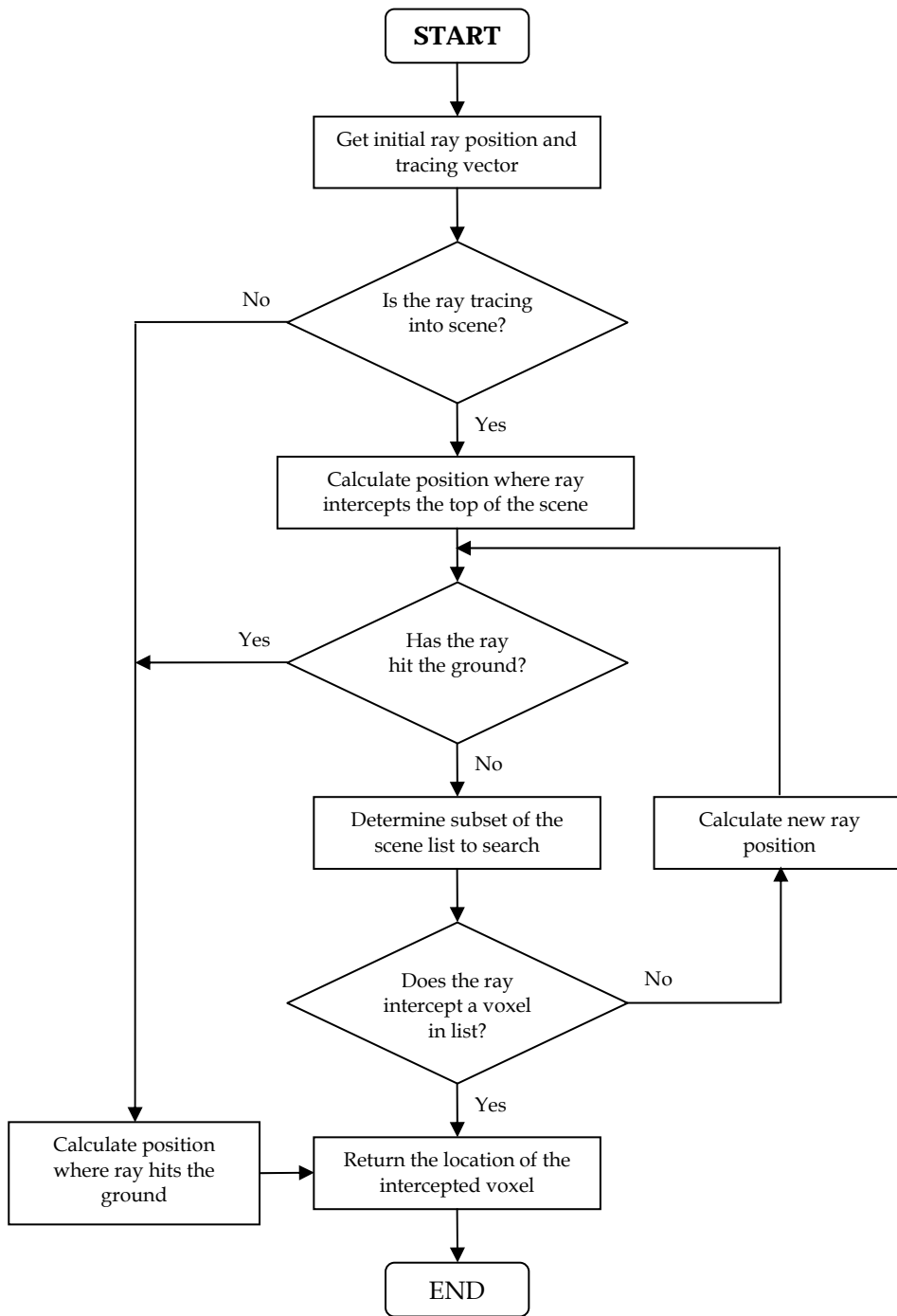


Figure 20: Flow chart of the ray tracing algorithm

8. Acknowledgements

The author would like to thank Dr. David Mewett for his input towards the development of the software tool and preparation of this report.

9. References

1. Aull, B. & Marino, R. (2005) "Three-dimensional imaging with arrays of Geiger-mode avalanche photodiodes", *Active and Passive Optical Components for WDM Communications V*, Proceedings of SPIE, vol. 6014.
2. Schilling, B.W.; Barr, D.N.; Templeton, G.C.; Mizerka, L.J.; Trussell, C.W. (2002) "Three-dimensional imaging of obscured targets by multiple-return laser radar", 23rd Army Science Conference, Orlando, Florida, 2-5 Dec 2002, paper JO-01.
3. Marino, R.M.; Davis, W.R.; Rich, G.C.; McLaughlin, J.L.; Lee, E.I.; Stanley, B.M.; Burnside, J.W.; Rowe, G.S.; Hatch, R.E.; Square, T.E.; Skelly, L.J.; O'Brien, M.; Vasile, A.; Heinrichs, R.M. (2005) "High-resolution 3D imaging laser radar flight test experiments", *Laser Radar Technology and Applications X*, Proceedings of SPIE, vol. 5791, pp. 138-151.

| DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION DOCUMENT CONTROL DATA | | | | 1. PRIVACY MARKING/CAVEAT (OF DOCUMENT) | |
|---|--------------------------|---|--|---|------------------------------|
| 2. TITLE Design of a Foliage Penetrating LADAR Simulation Tool | | 3. SECURITY CLASSIFICATION (FOR UNCLASSIFIED REPORTS THAT ARE LIMITED RELEASE USE (L) NEXT TO DOCUMENT CLASSIFICATION) Document (U) Title (U) Abstract (U) | | | |
| 4. AUTHOR(S) Mark Graham | | 5. CORPORATE AUTHOR DSTO Defence Science and Technology Organisation PO Box 1500 Edinburgh South Australia 5111 Australia | | | |
| 6a. DSTO NUMBER DSTO-GD-0577 | | 6b. AR NUMBER AR-014-521 | | 6c. TYPE OF REPORT General Document | 7. DOCUMENT DATE May 2009 |
| 8. FILE NUMBER 2009/1002636 | 9. TASK NUMBER 07/105 | 10. TASK SPONSOR Intelligence & Security Group | 11. NO. OF PAGES 33 | | 12. NO. OF REFERENCES 3 |
| 13. DOWNGRADING/DELIMITING INSTRUCTIONS To be reviewed three years after date of publication. | | | 14. RELEASE AUTHORITY Chief, Intelligence, Surveillance and Reconnaissance Division | | |
| 15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT <i>Approved for public release</i> OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SA 5111 | | | | | |
| 16. DELIBERATE ANNOUNCEMENT No Limitations | | | | | |
| 17. CITATION IN OTHER DOCUMENTS Yes | | | | | |
| 18. DSTO RESEARCH LIBRARY THESAURUS http://web-vic.dsto.defence.gov.au/workareas/library/resources/dsto_thesaurus.shtml Laser Radar, Computer programs, Simulation, Modelling | | | | | |
| 19. ABSTRACT A simulation tool was developed using MATLAB and its Graphical User Interface Development Environment (GUIDE) to simulate aspects of an airborne foliage-penetrating Laser Detection and Ranging (LADAR) system in scenarios designed to contribute towards the military operational use of such a system. In particular, the simulation tool is intended for conducting analysis on how best to task the aircraft and position the sensor. This document provides an overview of the graphical user interface and software including: the design challenges; dealing with the input scenery data; modelling the sensor and platform flight path; and planned analysis of the simulation results. | | | | | |